Everybody be cool, this is a robbery!

Jean-Baptiste Bédrune et Gabriel Campana
jean-baptiste.bedrune@ledger.fr
gabriel.campana@ledger.fr

Ledger Donjon

Résumé. Les HSM *(Hardware Security Modules)* sont des équipements électroniques utilisés comme brique cryptographique de confiance dans des environnements nécessitant de hautes exigences de sécurité. Ils sont utilisés comme enclave de sécurité afin de générer, stocker et protéger des clés cryptographiques. La protection de ces clés repose à la fois sur des mécanismes logiciels et matériels. Dans cet article, nous présentons la méthodologie que nous avons utilisée pour évaluer et améliorer la sécurité d'un modèle de HSM.

Un HSM n'étant pas un équipement courant, nous détaillons tout d'abord son rôle, son fonctionnement, ses interactions avec le monde extérieur et les types de messages qu'il traite. Notre méthodologie est ensuite expliquée ; elle a permis de découvrir plusieurs vulnérabilités sur cet équipement en quelques semaines. Certaines de ces vulnérabilités et leur exploitation sont par la suite présentées. Nous montrons plusieurs chemins d'attaque permettant à un attaquant non authentifié de prendre le contrôle total du HSM. Cela rend ensuite possibles la récupération de tous les secrets du HSM (que ce soit les clés cryptographiques ou les authentifiants des administrateurs), et l'introduction d'une porte dérobée persistante, survivant à une mise à jour complète du firmware.

Tous les problèmes découverts ont été divulgués au fabricant de manière responsable, lui laissant suffisamment de temps pour publier des mises à jour de son firmware contenant des correctifs de sécurité. Nous montrons enfin comment réduire grandement la surface d'attaque, en implémentant un module de filtrage des requêtes dans le HSM. L'introduction de ce module empêche d'atteindre, par construction, la plupart des problèmes que nous avions identifiés.

1 Introduction

1.1 Qu'est-ce qu'un HSM ?

Un HSM est un élément matériel, généralement une carte PCI ou une *appliance* réseau, composé notamment d'un processeur accompagné d'un ou plusieurs cryptoprocesseurs (accélérateurs matériels dédiés aux calculs cryptographiques). Des dispositifs matériels assurent une protection contre les tentatives d'intrusion et d'altération physiques.

Les HSM sont utilisés comme une enclave de sécurité pour stocker et effectuer des calculs sur des données sensibles. À l'instar des cartes à puce, un HSM génère et manipule lui-même des clés de chiffrement, offrant ainsi la possibilité d'effectuer des opérations cryptographiques tout en gardant ces clés confinées. Les infrastructures à clés publiques font par exemple usage des HSM pour générer et stocker les clés privées des autorités de confiance, et signer les certificats générés. Une utilisation courante en milieu bancaire est la génération, le stockage et la gestion des clés publiques et privées de l'infrastructure à clés publiques. Dans certains cas les transactions effectuées par les cartes bancaires sont aussi validées par ces HSM.

Des API cryptographiques telles que PKCS #11 [6] ou CryptoAPI sont implémentées pour assurer la communication avec le HSM et son utilisation par des logiciels tiers.

Les HSM peuvent respecter des standards de sécurité tels que FIPS ou les critères communs, qui définissent notamment des normes relatives aux algorithmes cryptographiques ou à la sécurité physique des composants sensibles. Les attaques logicielles ne semblent cependant pas suffisamment prises en compte ; nous verrons dans la suite que les implémentations doivent être particulièrement robustes pour assurer la sécurité des secrets contenus dans le HSM.

1.2 Brève introduction à PKCS #11

PKCS #11 est un standard qui définit une interface générique pour dialoguer avec un périphérique cryptographique. Un périphérique cryptographique est un matériel capable d'effectuer des opérations cryptographiques, comme un HSM ou une carte à puce. Le standard a été élaboré par RSA Laboratories, en relation avec d'autres entreprises et des institutions et est, depuis 2013, maintenu et mis à jour par l'OASIS PKCS11 Technical Committee. Il définit une API appelée *Cryptoki* (*Cryptographic Token Interface*) permettant de manipuler des objets cryptographiques courants (clés publiques, privées ou secrètes, certificats, etc.) et d'effectuer des opérations (chiffrement, signature, vérification de signature, dérivation de clés, etc.) sur ceux-ci.

L'API est générique dans le sens où elle est indépendante de la plateforme sur laquelle elle est exécutée, et indépendante du périphérique cryptographique.

Tokens et slots *Cryptoki* offre une vue logique des périphériques : chaque périphérique est appelé *token*. L'interface physique pour communiquer

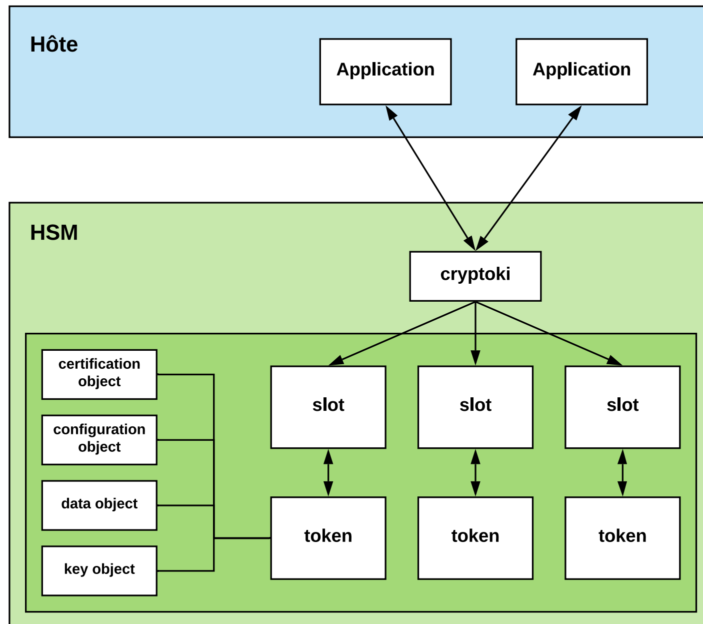

Fig. 1. Utilisation de *Cryptoki* par les applications

avec les *tokens* est appelée *slot*. Par exemple, une carte à puce est un *token*, et le lecteur de carte un *slot*. Dans le cas d'un HSM, cette distinction est plus délicate, car la séparation entre un *token* et un *slot* est une séparation logique. Les *slots* permettent d'accéder à des fonctionnalités différentes : il peut exister, par exemple, un *slot* dédié aux fonctions d'administration du HSM. Le *token* est le biais par lequel les opérations sont réalisées sur ce *slot*.

Lorsqu'un utilisateur se connecte à un *token*, il crée une *session*. Cette session peut être authentifiée ou non. Dans le second cas, en plus de certaines spécificités liées au périphérique, la session aura accès aux objets marqués comme privés.

Objets Que sont ces objets ? Les objets définis par PKCS #11 sont divisés en classes : ce sont soit des clés (secrètes, publiques et privées), soit des certificats, comme des certificats X.509, soit des données, comme des paramètres de domaine pour DSA ou ECDSA. Ces objets possèdent des *attributs* définissant leur comportement et les actions réalisables sur ceux-ci.

Ces attributs sont cruciaux pour la sécurité : par exemple, assigner l'attribut *SENSITIVE* à un objet signifie que cet objet est sensible, et que sa valeur ne peut pas être lue. C'est un attribut fréquemment utilisé pour les clés secrètes. Donner la valeur `false` à l'attribut *EXTRACTABLE* empêchera toute extraction de l'objet depuis le HSM, même wrappé par une autre clé. Enfin, les objets ayant l'attribut *PRIVATE* sont marqués comme privés et ne sont accessibles qu'après authentification.

Mécanismes Les mécanismes définissent précisément la manière d'exécuter une opération cryptographique. Par exemple, *Cryptoki* possède une fonction pour chiffrer des données. Le mécanisme est un paramètre de cette fonction, indiquant quelles primitives utiliser. `CKM_AES_CBC_PAD` signifie que les données doivent être chiffrées avec la primitive de chiffrement AES, avec le mode opératoire CBC, et le remplissage PKCS #8.

Il existe des mécanismes pour chiffrer, déchiffrer, hacher, signer et vérifier des données, dériver ou wrapper des clés de chiffrement, etc. Chaque périphérique implémente ou non un mécanisme. Un HSM va typiquement offrir beaucoup plus de mécanismes qu'une carte à puce.

Rôles Deux types d'utilisateurs sont définis dans PKCS #11 : les officiers de sécurité (SO) et les utilisateurs normaux. Seuls les utilisateurs normaux, une fois authentifiés, ont le droit d'accéder aux objets privés d'un *token*.

Le rôle du SO est d'initialiser un *token* et de définir les codes PIN des autres utilisateurs. Il peut également accéder à certains objets publics. Le niveau de privilèges du SO est celui le plus élevé : même s'il ne peut pas directement accéder aux objets privés, il peut modifier l'authentifiant de l'utilisateur associé au *token*.

1.3 État de l'art

L'état de l'art sur la sécurité des HSM n'est pas très riche et traite en grande majorité de la sécurité de PKCS #11. Cette section liste les différents types de vulnérabilités qu'on trouve aujourd'hui dans la littérature.

Problèmes intrinsèques à PKCS #11 La plupart des recherches se concentrent sur la sécurité de PKCS #11, et non sur celle d'une mise en œuvre particulière. Les articles et les présentations de J. Cluclow [4] et G. Steel [9] offrent une bonne vue d'ensemble des attaques sur la spécification. Nous recommandons également la lecture du livre blanc de Cryptosense [8].

Une attaque classique porte sur une clé ayant les droits `CKA_WRAP` et `CKA_DECRYPT`. Supposons qu'il existe une telle clé k_1, ainsi qu'une seconde clé k_2 marquée comme *SENSITIVE* et *EXTRACTABLE*. Cette dernière clé ne devrait jamais être accessible en clair.

Pourtant, un attaquant peut demander de *wrapper* k_2 avec k_1, pour obtenir une version chiffrée de k_2, puis déchiffrer le résultat toujours avec k_1. Le *token* renverra la clé en clair.

De tels problèmes ont été traités dans un article (cf. [2]) présenté à SSTIC en 2014, qui détaille la création d'un moteur de filtrage personnalisable pour PKCS #11.

Mécanismes faibles Gemini, un site d'échanges de cryptomonnaies, a publié sur son blog deux vulnérabilités touchant les HSM Luna de SafeNet, toutes deux liées à l'existence de mécanismes standard de PKCS #11 et aboutissant à l'extraction des clés sur le HSM par un utilisateur authentifié [7].

Le premier mécanisme est `CKM_EXTRACT_KEY_FROM_KEY`. Il crée une clé secrète à partir des bits d'une clé secrète existante. L'attaque consiste à extraire des bits d'une clé secrète de type AES-256 pour créer plusieurs petites clés secrètes, de 16 bits par exemple. Ces nouvelles clés peuvent être utilisées comme des clés HMAC, car il n'y a pas de prérequis sur la taille de ces clés. Un attaquant peut ensuite signer des messages avec chacune des petites clés HMAC créées, bruteforcer chaque clé HMAC et en déduire la valeur complète de la clé d'origine.

Le second mécanisme est `CKM_XOR_BASE_AND_DATA`, dérivant une nouvelle clé à partir d'une clé existante et d'une donnée choisie. Si une donnée courte est spécifiée, la clé résultante aura la taille de cette donnée. En utilisant cette nouvelle clé comme une clé HMAC, encore une fois, il est possible de calculer complètement la clé d'origine octet par octet.

Toutes les clés possédant l'attribut `CKA_DERIVE` (indiquant qu'un objet peut servir à dériver une clé) ou `CKA_MODIFIABLE` peuvent être extraites avec ces méthodes, aussi bien les clés secrètes que privées.

Ces problèmes ne sont pas propres à SafeNet, mais bien aux spécifications de PKCS #11. La même attaque avait par ailleurs déjà été présentée à CHES douze ans plus tôt [4]. La présentation en question était centrée sur les spécifications de PKCS #11, sans cibler de matériel ou d'implémentation particulière. L'auteur proposant d'extraire 40 bits d'une clé afin de l'utiliser en tant que clé RC4. Il montrait également d'autres attaques, toutes respectant le protocole PKCS #11.

SafeNet a pris en compte ces problèmes en désactivant dans la configuration par défaut les mécanismes faibles susceptibles d'aboutir à de tels scénarios. Il s'agit des mécanismes ajoutant des données, extrayant des octets ou effectuant des ou exclusifs avec une clé existante.

Problèmes spécifiques Enfin, il existe des vulnérabilités spécifiques à la mise en œuvre de *Cryptoki*. Il peut s'agir d'une non-conformité d'une primitive cryptographique entraînant une vulnérabilité (attaque par courbe invalide sur les HSM Utimaco [10], faiblesse dans la génération de clé [5], notamment), d'une corruption mémoire, d'un problème de logique, etc.

C'est ce type de problème sur lequel nous nous concentrons dans cette étude : nous n'avons pas évalué la conformité de l'interface *Cryptoki* du HSM.

2 Étude préliminaire et outillage

Le HSM évalué est une carte PCI Express. Il existe également sous forme de matériel applicatif réseau ; nous supposons que le fonctionnement interne est très similaire, mais nous n'avons pas pu le vérifier car nous ne possédons pas ce modèle. Ce HSM est certifié FIPS 140-2 niveau 3.

Nous présentons dans cette section comment nous avons pu accéder au firmware du HSM et comment nous l'avons modifié afin d'installer des utilitaires facilitant son analyse (`gdbserver`, notamment).

2.1 Présentation du HSM

La documentation sur l'implémentation matérielle du HSM est assez laconique. Notre analyse nous a permis de comprendre comment il était réalisé. Il se présente sous la forme d'une carte PCIe dont les composants sont recouverts par une couche de résine époxy. Il possède un processeur de type PowerPC, conçu spécialement pour le fabricant, lequel exécute un système Linux minimaliste.

Un port série et un port USB sont présents sur la carte PCI, dans l'éventualité ou un lecteur de carte à puce serait utilisé. Un contrôleur Ethernet est aussi visible, bien que le connecteur associé ne soit pas présent.

Nous supposons que la version réseau de ce HSM, que nous ne possédons pas, diffère matériellement de la version PCIe étudiée dans cet article uniquement par la présence de ce connecteur Ethernet. L'appliance dans laquelle elle est préinstallée, d'après la documentation du constructeur,

Fig. 2. Exemple de carte PCI Express

est un serveur Linux classique équipé d'un processeur Intel E5500 avec 4Go de RAM et d'un disque dur de 500 Go.

Afin d'autoriser la communication entre la machine hôte et la carte PCI et assurer son administration, des modules noyau ainsi que des logiciels graphiques et en ligne de commande pour la majorité des systèmes d'exploitation sont fournis par le constructeur. L'étude a été réalisée avec un hôte sous Linux, mais cela n'a *a priori* pas d'influence sur le comportement du HSM.

Un SDK (*Software Development Kit*) est aussi fourni pour développer des modules interagissant avec le HSM. Cette fonctionnalité est détaillée par la suite. Par *module*, nous entendons deux types de programmes : des outils externes interrogeant le HSM, et également des modules natifs exécutés sur le HSM.

Administration Le constructeur du HSM fournit 2 rôles spécifiques à l'administration du HSM, additionnels à ceux déjà définis par le standard PKCS #11 : officier de sécurité administrateur (ASO) et administrateur, associés à l'unique slot `admin`. De façon similaire aux rôles SO et utilisateur, l'ASO crée et gère le rôle administrateur. L'administrateur est le seul rôle possédant les privilèges pour configurer des paramètres système n'étant pas liés à un token, comme la valeur de l'horloge temps réel, la configuration des fichiers de journalisation, la création et la suppression de slots et de tokens, ou encore la mise à jour du firmware.

Chaque slot, à l'exception du slot `admin`, est créé par l'administrateur et initialisé automatiquement avec le token de l'utilisateur unique associé au slot. Par défaut, un seul slot utilisateur est disponible. L'administrateur peut ajouter des slots à volonté, mais les performances du HSM diminuent en fonction du nombre de slots créés.

Le slot `admin` est prévu pour l'administrateur et utilisé pour la configuration et l'administration du HSM. Il y a un seul administrateur par

HSM. L'administrateur possède le token admin, qui contient les objets d'administration. Ces objets d'administration représentent le matériel et contiennent les paramètres de configuration du HSM ; certains de ces objets sont accessibles en lecture et/ou en écriture par l'administrateur. Ils sont automatiquement créés lors de l'initialisation du HSM.

Les PIN sont les authentifiants des utilisateurs. Ce sont des chaînes de caractères ANSI sensibles à la casse, d'une longueur de 1 à 32 caractères. Deux options de configuration empêchent le bruteforce du PIN : une suspension du compte après un nombre de tentatives de connexion avec un PIN invalide, et une augmentation du délai de connexion après chaque tentative. Ces règles sont définies par l'ASO.

Communication avec l'hôte La version réseau, que nous n'avons pas testée, communique par Ethernet. Le standard `PCI-E` utilisé par la carte PCIe est implémenté par 2 modules noyau développés par le constructeur, l'un pour le système hôte et l'autre pour le HSM. Ces modules noyau assurent par ailleurs la transmission des messages échangés entre l'hôte et le HSM aux processus userland responsables de leur traitement.

Sur le HSM, plusieurs interfaces sont accessibles via ce mode de communication : *Cryptoki* pour toutes les opérations cryptographiques PKCS #11, ou encore un canal d'administration.

Stockage persistant Toutes les données persistantes sont stockées dans une mémoire flash de 64 Mo. On trouve parmi ces données l'image Linux qui va être chargée lors du démarrage, le firmware personnalisé, s'il est présent (cf. plus bas), les logs et les objets PKCS #11.

Nous nous intéressons ici aux objets PKCS #11. Ils sont enregistrés dans une partition spécifique, sur un système de fichiers propriétaire. Chaque objet est composé d'une liste d'attributs. Par exemple, une clé cryptographique possède notamment un attribut spécifiant son type, d'autres attributs spécifient sa valeur, son rôle, et un autre indique si elle est exportable, etc. Un objet est stocké en flash comme une séquence d'attributs sérialisés. Le format est un format binaire classique : type, longueur, valeur optionnelle.

L'implémentation distingue deux types d'attributs : les attributs *sensibles* et les autres. Ces derniers sont enregistrés en clair dans la flash. Les autres sont chiffrés. Voici les attributs d'une clé privée ECDSA :

```
{
    CKA_LABEL = "ECDSA Private Key",
    CKA_CLASS = CKO_PRIVATE_KEY,
    CKA_KEY_TYPE = CKK_EC,
    CKA_SENSITIVE = false,
    CKA_SIGN = true,
    CKA_ECDSA_PARAMS = secp192r1,
    CKA_VALUE = 20 47 F1 91 64 45 2D 8F ... (0x208 bytes)
    CKA_ALWAYS_SENSITIVE = false,
    CKA_VALUE_LEN = 0x200,
    CKA_ID = "2018083112385000",
    ...
}
```

Listing 1. Attributs d'une clé privée ECDSA

On peut observer que l'identifiant de la clé, ses paramètres de domaines, etc. sont en clair, et que la valeur de la clé est chiffrée. Les attributs sensibles d'un objet sont en effet chiffrés et déchiffrés à la volée par la couche PKCS #11 (*Cryptoki*), sans qu'un utilisateur ait besoin de fournir un secret spécifique. La couche de chiffrement ne protège pas les éléments à chaud : le but est d'empêcher une attaque physique. La clé de chiffrement des attributs sensibles est écrite dans une mémoire externe, qui sera effacée si des capteurs détectent une anomalie. Si le HSM est déplacé ou ouvert, tous les secrets deviennent immédiatement inaccessibles.

Nous déduisons de cette observation qu'il n'y a qu'une séparation logique entre les différents slots du HSM : les objets de chacun des slots sont tous enregistrés dans la même partition de flash, et les secrets sont protégés par la même clé, initialisée lors du formatage de la flash. La rétro-ingénierie du firmware montre que cette clé est totalement indépendante des données d'authentification des utilisateurs (identifiant, code PIN).

L'obtention d'une exécution de code sur un des slots du HSM est a priori suffisante pour récupérer l'intégralité des secrets de chacun des slots.

2.2 Analyse du firmware

Un CD-ROM est fourni avec la carte. Il contient :

— des logiciels graphiques et en ligne de commande pour administrer le HSM, pour divers systèmes d'exploitation, dont Windows et Linux ;
— un SDK (*Software Development Kit*) ;
— des exemples d'utilisation de l'API PKCS #11 dans le langage C ;
— diverses documentations destinées aussi bien aux développeurs qu'aux administrateurs ;
— un firmware pour mettre à jour le HSM.

La documentation fournie par le constructeur offre une vision fonctionnelle et haut niveau de la solution. Analyser le fonctionnement du logiciel exécuté sur le HSM permet d'évaluer techniquement la mise en oeuvre de ces fonctionnalités et d'avoir une meilleure idée sur la surface d'attaque réelle. L'analyse des fichiers utilisés lors de la mise à jour du firmware du HSM montre que celui-ci est signé, mais n'est pas chiffré. Son analyse est donc facilitée.

Image du firmware Le firmware est un *devicetree* binaire signé. Les quatre premiers octets du firmware indiquent la taille complète du *devicetree*. Vient ensuite le *devicetree*, suivi de la signature de celui-ci. Le *devicetree* une fois extrait dans un nouveau fichier est lisible avec des outils standard :

```
$ dtc -I dtb -O dts -o fw.dt fw.dtb
/dts-v1/;

/ {
    timestamp = <0x5bd3670b>;
    description = "Kernel, initramfs and FDT blob";

    images {
        kernel {
            description = "vmlinux";
            data = [1f 8b 08 08 ee 66 d3 5b 02 03 76 ...
            type = "kernel";
            arch = "ppc";
            os = "linux";
            compression = "gzip";
        };

        ramdisk {
            description = "initramfs.cpio";
            data = [1f 8b 08 00 0a 67 d3 5b 00 03 ec ...
            type = "ramdisk";
            arch = "ppc";
            os = "linux";
            compression = "gzip";
        };

        fdt {
            description = "fdt";
            data = <0xd00dfeed 0x1e54 0x38 0x1774 0x28 ...
            type = "flat_dt";
            arch = "ppc";
            compression = "none";
        };
    };
    ...
};
```

Listing 2. Structure du firmware

Il contient donc un noyau très ancien, la version 2.26.8.8 datant de mars 2009, un ramdisk et un nouveau *devicetree* décrivant le matériel.

L'architecture indiquée est bien PowerPC. Le ramdisk est minimal : il comprend quelques bibliothèques et un exécutable assurant toutes les fonctionnalités du HSM, ainsi que quelques modules noyau pour gérer le matériel. Le noyau Linux semble compilé avec un minimum d'options. En revanche, aucun binaire n'est *strippé* et les symboles sont majoritairement présents.

Les fichiers de configuration, les chaînes de caractères trouvées ainsi que les versions des bibliothèques laissent penser que certaines parties datent de plus de 10 ans.

Émulateur Le SDK fourni par le fabricant contient un émulateur destiné à l'architecture x86. Les binaires ne sont pas non plus strippés et possèdent certaines fonctions présentes dans le firmware, lui destiné à un processeur PowerPC. Les outils de reverse engineering étant plus riches sur x86, il est possible, pour gagner du temps, d'étudier ces binaires plutôt que les versions présentes sur le firmware. Seules les fonctionnalités spécifiques au firmware, non présentes dans le SDK, devront être reversées à partir du firmware PowerPC.

2.3 Modules et développement d'outils

Ce HSM présente une fonctionnalité peu répandue : la possibilité d'ajouter des fonctionnalités à la volée au système exécuté sur le HSM.

Le constructeur fournit un SDK à cet effet. La chaîne de compilation produit un binaire pouvant être chargé sur le HSM par l'administrateur au travers de commandes système exécutées sur la machine hôte. L'API fournie offre la possibilité de hooker les fonctions PKCS #11 ainsi que de définir des *handlers* spécifiques sur certains types de messages reçus par le HSM.

L'analyse du binaire produit par la chaîne de compilation montre que c'est une simple bibliothèque ELF pour PowerPC. Il est alors envisageable de créer une bibliothèque ELF sans utiliser le SDK et d'exécuter un code arbitraire sur le HSM. Quelques tests montrent que ce code n'est pas isolé du processus ayant chargé la bibliothèque avec `dlopen`.

Des outils ont été développés en utilisant ces modules.

Shell Le premier module développé a pour but l'exécution de commandes arbitraires sur le HSM. Il est divisé en 2 parties distinctes : un client exécuté sur le système hôte prend en argument les commandes à exécuter et les transmet au module en cours d'exécution sur le HSM.

Le client (x86) et le module lui-même (PowerPC) utilisent les fonctions fournies par le SDK pour communiquer. Le module déclare un nouvel *handler* interprétant les messages envoyés par le client et implémente quelques primitives basiques : lecture, écriture et exécution de fichiers. Le SDK ne fournissant pas de bibliothèque C, les fonctions requises (par exemple `open`, `execve`, etc.) ont été développées pour effectuer directement des appels système.

Le client déployé sur l'hôte envoie au module un exécutable statique `busybox` cross-compilé pour PowerPC. Le module l'écrit sur le système de fichiers et il devient alors possible d'appeler `busybox`, depuis le système hôte, pour exécuter des commandes Linux standard sur le HSM, comme le montre la figure 3.

```
user@host:~$ ./module-shell --init
[*] uploading busybox-powerpc to /sbin/busybox
[*] creating symlinks (might take a few seconds)

user@host:~$ ./module-shell id
uid=0 gid=0

user@host:~$ ./module-shell ps fauxwww
PID   USER     TIME   COMMAND
    1 0           0:00 /init
    2 0           0:00 [kthreadd]
...
  728 0           0:00 /sbin/powerpc-4xx-pcscd
  730 0           0:00 /crashdump /HSM
  731 0           0:06 /HSM
 1086 0           0:00 /sbin/busybox ps fauxwww
```

Listing 3. Processus en cours d'exécution

L'obtention du shell confirme que le processus `HSM` tourne avec les privilèges *root* et donne la possibilité d'obtenir des informations qui ne sont pas disponibles par une analyse statique du firmware, comme la révision exacte du processeur.

Comme nous allons le voir dans la suite, ce shell établit les bases pour effectuer une analyse dynamique du système en cours, ce qui facilitera grandement la confirmation ou l'infirmation d'hypothèses faites par reverse engineering ainsi que le développement d'exploits.

Débogueur Le module précédent est utilisé pour uploader un exécutable statique `gdbserver` pour l'architecture PowerPC. Il n'est cependant pas possible de déboguer directement le processus `HSM` puisqu'il est responsable de la communication entre la carte PCI et le système hôte. Si son exécution est temporairement arrêtée à cause d'un point d'arrêt par exemple, son exécution sera interrompue définitivement puisque les paquets suivants ne pourront être traités.

Un canal de communication différent doit alors être envisagé, sans recours aux fonctionnalités offertes par le SDK qui reposent toutes sur le même mécanisme. Il aurait été possible d'utiliser des fonctionnalités matérielles présentes sur la carte PCI (comme le port USB et le port série, ou l'ajout d'un connecteur Ethernet), mais nous avons préféré choisir une approche purement logicielle.

Des canaux auxiliaires de communications doivent donc être trouvés. L'analyse dynamique du système révèle les primitives suivantes :

- Une zone de mémoire partagée semble inutilisée. Elle est accessible en lecture et en écriture par l'hôte, et uniquement en lecture par le HSM ; elle est utilisée pour transmettre des données depuis l'hôte vers le HSM ;
- L'analyse des fichiers de journalisation depuis l'hôte montre que le module a la possibilité d'enregistrer des messages de journalisation. Le HSM peut donc transmettre des données vers l'hôte par cette fonctionnalité, les messages étant habituellement lus avec la commande `hsmlog`.

En détournant ces mécanismes, la transmission de données est donc possible depuis l'hôte vers le HSM et inversement. Un système de messages rudimentaires contenant un identifiant unique et nécessitant un acquittement du pair est implémenté. Sans mécanisme de notification, il est aussi indispensable de vérifier à intervalle régulier si de nouveaux messages doivent être traités, l'usage du CPU est donc intensif sur l'hôte et le HSM. Enfin, les messages de journalisation n'acceptent qu'un ensemble restreint de caractères, et les messages doivent donc être encodés. Le système de communication n'est donc pas entièrement stable du fait de ces différentes contraintes, mais reste suffisamment performant pour les besoins de l'analyse.

Un programme reposant sur ce mécanisme est développé pour transmettre la sortie standard d'un programme exécuté sur le HSM vers l'hôte, et l'entrée standard de l'hôte vers le HSM. Cela rend possible la communication entre un client `gdb` sur l'hôte (x86) et un serveur `gdbserver` sur le module ; et finalement le débogage du processus `HSM`.

Notons que le processus `HSM` est initialement surveillé par un processus parent, `crashdump`, via `ptrace`. Dans l'éventualité où le processus `HSM` crashe, suite à un bug par exemple, `crashdump` envoie quelques informations de débogage à l'hôte avant de redémarrer le HSM. Si le processus `crashdump` se termine, le HSM redémarre aussi. Afin de pouvoir déboguer le processus `HSM`, un shellcode est injecté dans le tas du processus `crashdump` (qui est exécutable) pour effectuer un appel à `ptrace(PTRACE_DETACH)`.

2.4 Informations récupérées

Cette étude préliminaire accompagnée du développement de quelques outils donne les éléments nécessaires à la compréhension complète du fonctionnement du HSM.

L'intégralité de la flash est accessible en lecture grâce aux fichiers de périphériques `/dev/mtd`. Le système de démarrage est analysé et montre que le bootloader est une version légèrement modifiée d'`U-Boot`, sans mécanisme de *secure boot*.

La connaissance des processus en cours d'exécution et des modules du noyau utilisés détermine clairement la surface d'attaque accessible et ouvre la voie à la recherche de vulnérabilités de façon statique et dynamique.

3 Recherche de vulnérabilités et exploitation

Menaces Les menaces envisagées sont les suivantes, la liste n'étant pas exhaustive :

— Un attaquant non authentifié accède à des objets privés ;
— Un attaquant récupère des clés marquées comme non extractibles ;
— Un attaquant authentifié réussit à obtenir des droits normalement réservés au SO.

Nous faisons l'hypothèse que l'attaquant est capable d'exécuter des commandes sur la machine hébergeant le HSM. Son niveau de privilèges l'autorise à interagir avec le HSM. De façon plus technique, la communication entre les outils d'administration et le module noyau dialoguant avec le HSM est effectuée par le biais d'*ioctls* sur un *device* virtuel. L'attaquant a les permissions requises pour accéder à ce device en lecture et en écriture.

Enfin, nous postulons que le SO et l'ASO ne sont jamais malveillants, et ne considérons pas les chemins d'attaque nécessitant un accès physique.

3.1 Surface d'attaque

Les points d'entrée envisagés sont les suivants :

- L'hôte communique avec la carte PCIe via une mémoire partagée (*dual-ported RAM*). Nous étudions la robustesse de l'implémentation sur le HSM ;
- Le SDK fourni par le constructeur permet d'effectuer des appels à l'interface PKCS #11. Le traitement de ces appels est analysé ;
- La qualité des algorithmes cryptographiques implémentés est aussi étudiée.

Interfaces cryptographiques Le nombre de fonctions exposées par *Cryptoki* n'est pas si élevé : il y en a un peu plus de 70. Ce faible nombre est trompeur, et cache la complexité du code sous-jacent. Par exemple, la fonction d'initialisation d'un chiffrement, `C_EncryptInit`, prend un mécanisme en paramètre. Lors de l'instanciation du chiffrement, le HSM va récupérer l'objet *Cipher* associé à ce mécanisme et les opérations de chiffrement se feront avec cet objet. Le HSM expose 250 mécanismes, certains standards, d'autres propriétaires.

Chaque objet *Cipher* expose jusqu'à 24 fonctions, dont une grande partie est accessible depuis l'interface *Cryptoki*. Le même principe s'applique aux objets *Hash* et *PublicCipher*. Le nombre de fonctions réellement accessibles depuis l'API PKCS #11 est donc en réalité beaucoup plus important qu'on pourrait imaginer de prime abord.

Sérialisation des données Chaque appel à *Cryptoki* depuis le système hôte est transmis à la carte PCIe au moyen d'un driver noyau. La bibliothèque fournie dans le SDK sérialise les données à transmettre au noyau, lequel les transmet au HSM qui déserialise ces données avant de les traiter.

En observant les données sérialisées, on peut voir qu'elles contiennent les différents paramètres de la fonction appelée. Parmi ces paramètres, les plus complexes à décoder sont les attributs et les paramètres des mécanismes. Les attributs sont manipulés côté client comme un tableau de structures TLV :

```
CK_ATTRIBUTE pubAttr[] =
{
    {CKA_LABEL,          NULL,        0},    /* We will fill this in
        later. */
    {CKA_EC_PARAMS,      NULL,        0},    /* We will fill this in
        later. */
    {CKA_CLASS,          &pubClass,   sizeof(pubClass)},
    {CKA_KEY_TYPE,       &keyType,    sizeof(keyType)},
```

```
    {CKA_TOKEN,           &ckTrue,     sizeof(ckTrue)},
    {CKA_SENSITIVE,       &ckFalse,    sizeof(ckFalse)},
    {CKA_VERIFY,          &ckTrue,     sizeof(ckTrue)}
};
```

Listing 4. Attributs PKCS #11

Ces données peuvent contenir des objets complexes. `CKA_EC_PARAMS` contient, par exemple, les paramètres de domaines à utiliser pour une signature ECDSA. Ces paramètres sont sérialisés au format DER, ce qui signifie que le HSM intègre un décodeur ASN.1, format à la source de nombreuses vulnérabilités.

Le HSM traite 61 attributs standards et 72 attributs propriétaires. Leur type est varié : entiers, chaînes de caractères, suite d'octets, booléens, etc. La désérialisation doit gérer tous ces types correctement. Enfin, les fonctions recevant ces attributs doivent s'assurer qu'ils sont tous présents et que leur format et leur valeur sont valides. Cela ne semble a priori pas simple.

Les paramètres passés aux mécanismes sont également variés. Il s'agit parfois de structures, qui peuvent contenir un ou plusieurs pointeurs (cf. listing 5). Leur désérialisation par le HSM doit également être réalisée avec précaution.

```
typedef struct CK_BIP32_CHILD_DERIVE_PARAMS {
    CK_ATTRIBUTE_PTR pPublicKeyTemplate;
    CK_ULONG ulPublicKeyAttributeCount;
    CK_ATTRIBUTE_PTR pPrivateKeyTemplate;
    CK_ULONG ulPrivateKeyAttributeCount;
    CK_ULONG_PTR pulPath;
    CK_ULONG ulPathLen;
    CK_OBJECT_HANDLE hPublicKey; // output parameter
    CK_OBJECT_HANDLE hPrivateKey; // output parameter
    CK_ULONG ulPathErrorIndex; // output parameter
} CK_BIP32_CHILD_DERIVE_PARAMS;
```

Listing 5. Paramètres complexes désérialisés par le HSM

Cryptoki expose donc de nombreux mécanismes, manipule et désérialise en permanence des données contrôlables par un attaquant. Le module n'est compilé avec aucune des options de *hardening* et est exécuté en tant que *root*. Toutes ces conditions facilitent le travail d'un attaquant.

3.2 Première exécution de code

Un premier objectif lors de l'étude a été de rechercher une corruption mémoire facile à exploiter. Une méthode systématique simple a fonctionné : nous avons listé toutes les fonctions appelant `memcpy` en ne lui spécifiant pas une longueur constante, et dont le buffer de destination était sur la pile. Il y a un peu moins de 700 appels à `memcpy` dans `libcryptoki.so`, et très peu parmi ceux-ci valident les deux conditions. Nous les avons donc analysés.

Une seule fonction présentait a priori un *stack overflow* : `MilenageDerive`, utilisée par le mécanisme `CKM_MILENAGE_DERIVE`. Elle dérive des clés pour les fonctions *f3*, *f4*, *f5* et $f5^*$ de MILENAGE, ensemble d'algorithmes d'authentification UMTS.

La documentation indique que ce mécanisme requiert une clé AES de 16 octets initialisée sur le slot du HSM, ainsi qu'un diversifiant propre à l'opérateur, lui aussi de 16 octets. Or, la fonction de dérivation ne vérifie pas la longueur de ces éléments et les copie dans des tableaux de taille fixe sur la pile. L'utilisation d'une « grande » clé va entraîner un dépassement de pile classique. Comme aucun cookie de pile n'est présent, l'exécution de code arbitraire est immédiate.

```
CK_BYTE key_data[256] = {[0 ... 255] = 0xcc}; // wow such big key

rv = CreateSecretKey(hSession, "yolokey", 1, 1, CKK_GENERIC_SECRET,
                     key_data,  sizeof(key_data), &hSecretKey);
if (rv != CKR_OK) { return 0; }

CK_MILENAGE_DERIVE_PARAMS params;
params.f = 0x40;  // MILENAGE f4
params.hObject = hSecretKey;
for (int i = 0; i < 16; i++) { params.random[i] = i; }

CK_MECHANISM mechanism = {
  CKM_MILENAGE_DERIVE, &params, sizeof(params)};

CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_GENERIC_SECRET;

CK_ATTRIBUTE template[] = {{CKA_CLASS, &keyClass, sizeof(keyClass)},
                          {CKA_KEY_TYPE, &keyType, sizeof(keyType)}};

// L'appel de C_DeriveKey avec le émcanisme CKM_MILENAGE_DERIVE va
// appeler MilenageDerive et planter le HSM.
rv = C_DeriveKey(hSession, &mechanism, hSecretKey, template,
                 NUMITEMS(template), &hDerivedKey);
```

Listing 6. Plantage lors de la dérivation d'une clé MILENAGE

Ce premier test a validé l'absence d'un mécanisme de sécurité éventuel qui aurait empêché une exécution de code. Le bug est simple à déclencher, mais nécessite d'être authentifié sur le HSM, car il faut au préalable ajouter une clé secrète, et l'écriture d'un shellcode sur la pile rend difficile la reprise stable de l'exécution, d'autant plus si le shellcode à injecter est complexe. Nous détaillons dans les parties suivantes comment découvrir des vulnérabilités plus intéressantes, et créer un payload adapté à un système sans outil et sans moyen de communication usuel avec l'extérieur.

3.3 Fuzzing

L'intégralité des commandes supportées par le HSM est obtenue par le reverse engineering de la bibliothèque *Cryptoki*. Le standard PKCS #11 est implémenté, ainsi que quelques commandes supplémentaires. Certaines fonctions sont accessibles sans authentification, d'autres en tant que `User`, d'autres en tant qu'`Admin`. Les fonctions nécessitant le moins de privilèges sont ciblées en priorité.

Les exemples d'utilisation de l'API PKCS #11 fournis par la documentation couvrent toutes les commandes supportées. Un *fuzzer* rudimentaire a été écrit pour muter les données transférées depuis les bibliothèques userland au module noyau via des *hooks* sur les fonctions d'envoi de messages. Certaines précautions doivent être prises, car la robustesse du module noyau du système hôte n'est pas suffisante : des crashes peuvent avoir lieu lors du traitement de requêtes PKCS #11 invalides.

De même, la configuration du système d'exploitation du HSM est modifiée en utilisant la commande `sysctl` (par le biais du shell obtenu via un module) pour éviter les dénis de service dus à des situations où la mémoire disponible est insuffisante pour traiter les requêtes reçues.

Le processus `crashdump` précédemment décrit est attaché au processus principal `HSM` afin de déterminer si des crashes ont lieu. Le cas échéant, le HSM redémarre et une capture de l'état des registres est écrite dans les fichiers de journalisation. Cela permet d'effectuer manuellement un tri rapide des crashes obtenus.

14 vulnérabilités distinctes ont été découvertes et illustrent de manière presque exhaustive les *bugs* de corruption mémoire possibles. Elles concernent des composants variés du HSM et nous en présentons quelques-unes ici.

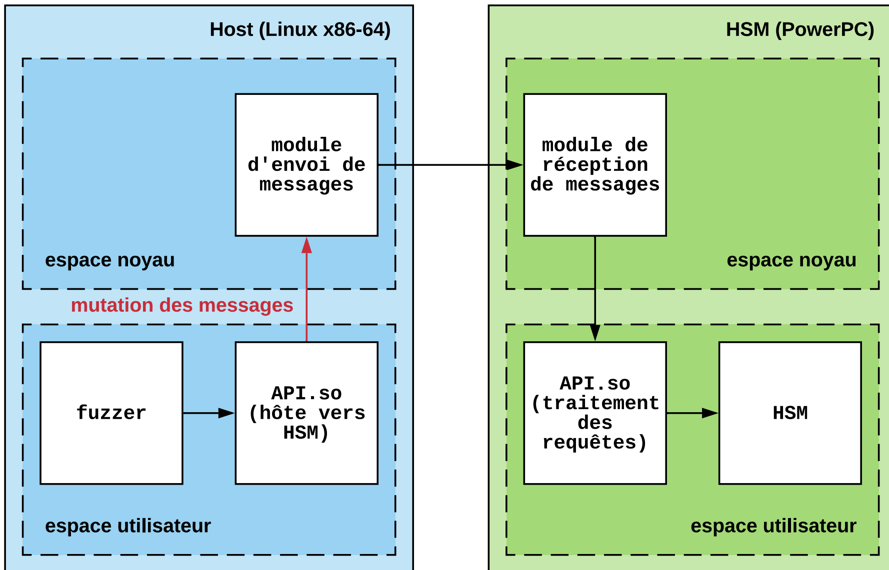

Fig. 3. Fuzzing

3.4 Fuite Mémoire

Une vulnérabilité similaire à Heartbleed est présente dans les fonctions PKCS #11 de gestion des attributs, permettant à un attaquant de lire partiellement la mémoire du tas du HSM. Sur l'hôte, la fonction de *Cryptoki* `C_CreateObject` crée de nouveaux objets et `C_SetAttributeValue` modifie la valeur d'un ou plusieurs attributs d'un objet :

```
CK_ATTRIBUTE certificateTemplate[] = {
    {CKA_CLASS, &certificateClass, sizeof(certificateClass)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_SUBJECT, subject, sizeof(subject)},
    {CKA_ID, id, sizeof(id)},
    {CKA_VALUE, certificateValue, sizeof(certificateValue)}
};
CK_ATTRIBUTE template[] = { { CKA_SUBJECT, "abc", 3 } };

C_CreateObject(hsSession, certificateTemplate, 5, &hCertificate);
C_SetAttributeValue(hsSession, hCertificate, template, 1);
```

Listing 7. Modification de la valeur d'un attribut

L'hôte sérialise les requêtes PKCS #11 `CreateObject` et `SetAttributeValue` afin de pouvoir les transmettre au HSM. La requête

`SetAttributeValue` sérialisée contient notamment la taille de la valeur de chaque attribut modifié. Le HSM ne s'assure pas lors de la désérialisation que la taille spécifiée correspond à la taille effective de la valeur : si la taille spécifiée est supérieure à la taille de la valeur de l'attribut donnée, l'attribut est initialisé avec de la mémoire du tas du HSM. `C_GetAttributeValue` peut finalement être appelé pour récupérer la valeur de l'attribut et donc la mémoire du HSM.

```
attacker@host:~$ ./heartbleed user $((0x78)) | hd \
  -e '"%08.8_Ax\n"' \
  -e '"%08.8_ax " 8/1 " %02x"' \
  -e '"  |" 8/1 "%_p" "|\n"'
[*] modifying buffer size to 0x78
00000000  62 6c 61 68 00 90 47 6c |blah..Gl|
00000008  00 00 00 04 01 00 00 00 |........|
00000010  01 00 00 00 01 00 00 00 |........|
00000018  01 01 01 00 00 01 01 00 |........|
00000020  00 00 08 01 73 75 62 6a |....subj|
00000028  65 63 74 00 00 00 01 02 |ect.....|
00000030  00 00 00 01 01 00 00 00 |........|
00000038  00 11 00 00 00 08 01 10 |........|
00000040  43 d8 5c 37 a7 57 6b 00 |C.\7.Wk.|
00000048  00 01 61 00 00 00 04 01 |..a.....|
00000050  00 00 00 01 80 00 01 02 |........|
00000058  00 00 00 10 01 32 30 31 |.....201|
00000060  38 30 39 30 33 30 38 30 |80903080|
00000068  33 34 39 30 30 00 00 01 |34900...|
00000070  0a 00 00 00 01 01 01 80 |........|
```

Listing 8. Hexdump de la mémoire du HSM extraite avec l'exploit de fuite mémoire

Certaines clés ne sont pas effacées en mémoire et il est possible de les retrouver avec un tel type de bug. On retrouve de même le code PIN des administrateurs en clair. Les adresses de certains objets sont aussi présentes dans les extraits de mémoire récupérés, facilitant potentiellement l'écriture d'exploits pour d'autres vulnérabilités.

3.5 Conformité des mécanismes cryptographiques

Il nous aurait paru surprenant d'identifier un problème dans la conformité des mécanismes cryptographiques mis en œuvre dans le HSM. Pourtant, une non-conformité sur les signatures PKCS #1 1.5 a été découverte.

Deux méthodes de recherche automatisée de non-conformités permettent de découvrir régulièrement des vulnérabilités dans les mises en œuvre d'algorithmes cryptographiques.

Comparaison des résultats de primitives cryptographiques sur différentes mises en œuvre Nous comparons les résultats produits à partir d'entrées valides ou mutées. Cette méthode a été présentée par Kudelski Security sous l'appellation « Crypto Differential Fuzzing » [1]. Notre méthode diffère légèrement, car nous instrumentons la mise en œuvre qui nous sert de référence avec libFuzzer ou AFL pour augmenter la couverture de code et cibler des chemins intéressants.

Validation des vecteurs de test fournis par Wycheproof Wycheproof [3] inclut des vecteurs de test pour quelques algorithmes (ECDH, ECDSA, RSA, AES notamment) afin de tester les limites de leur implémentation. Ces vecteurs testent des subtilités dans l'encodage ASN.1 des signatures, des cas limites pour certains algorithmes de signature, etc. Wycheproof a été écrit pour valider les fournisseurs cryptographiques Java. En plus des vecteurs de test permettant d'identifier certaines faiblesses, il permet également de détecter des biais lors de la génération d'éléments secrets, etc. La publication des vecteurs de test rend possible l'utilisation d'une partie de l'outil pour tester n'importe quelle bibliothèque.

Résultats Dans notre cas, peu de vecteurs de test sont utilisables, principalement car *Cryptoki* ne vérifie pas des signatures encodées en DER. Seuls les vecteurs AES et RSA ont été testés. Nous avons découvert que le padding des signatures RSA PKCS #1 v1.5 n'était pas correctement vérifié. Le message à signer doit être paddé ainsi, avec `PS` une suite d'octets valant tous `0xFF` et `T` le message à signer, précédé de l'identifiant de la fonction de hachage utilisée :

```
EM = 0x00 || 0x01 || PS || 0x00 || T.
```

`PS` n'est pas correctement vérifié. Chaque octet peut prendre n'importe quelle valeur, ce qui induit une vulnérabilité de type « forge existentielle » sur RSA : un attaquant est en mesure de créer une signature pour un message qu'il ne contrôle pas.

La présence de ce problème est étonnante, la conformité cryptographique étant une fonctionnalité clé (et basique) d'un HSM, d'autant plus que la vulnérabilité a été découverte de manière automatisée.

3.6 Confusion de type sur une fonction PKCS #11

Le fuzzing des fonctions PKCS #11 résulte en un crash du HSM lors de l'appel à `C_DigestUpdate`. L'analyse de la *stacktrace* montre que ce

crash est provoqué par un accès invalide à une adresse mémoire dans une fonction de mise à jour d'un contexte du condensat RIPEMD-128. Notons qu'aucune des fonctions PKCS #11 appelées ne requiert une authentification.

Analyse du crash Les différents appels aux fonctions PKCS #11 effectués par le fuzzer et résultant en un crash sont illustrés dans le listing 9.

```
CK_MECHANISM digestMechanism = { CKM_RIPEMD128 , NULL_PTR , 0 };
unsigned char state[4096] , data[32];
CK_ULONG ulStateLen;

C_DigestInit(hSession , &digestMechanism);
C_GetOperationState(hSessions[i], NULL_PTR , &ulStateLen);
C_GetOperationState(hSession , state , &ulStateLen);
mutate(state , ulStateLen);
C_SetOperationState(hSession , state , ulStateLen , 0, 0);
C_DigestUpdate(hSession , data , sizeof(data));
```

Listing 9. Fuzzing de fonctions de manipulation de condensat

`C_GetOperationState` récupère l'état cryptographique d'une session, tandis que `C_SetOperationState` le restaure. Le crash obtenu montre que le mécanisme de restauration est loin d'être robuste : un seul octet est modifié par le fuzzer dans la fonction `mutate`, représentant le type de condensat utilisé par la session (ici `CKM_RIPEMD128`).

La rétro-ingénierie des mécanismes de manipulation de condensat montre qu'il n'y a pas de vérification sur le type de condensat restauré, et donc que celui-ci peut ne pas correspondre au condensat sauvegardé originellement. Chaque condensat est représenté par un objet différent sur le HSM. La modification du type de condensat lors de la restauration provoque l'instanciation d'un nouvel objet du type spécifié, mais initialisé avec la structure de l'objet initialement sauvegardé. Cette opération entraîne donc une confusion de type.

Exploitation Une étude approfondie des objets associés aux différents types de condensats supportés montre que ce crash peut être transformé en une primitive plus intéressante qu'un déni de service par un attaquant. La restauration de certains types de condensat résulte en l'appel de la fonction `memcpy`, où la valeur et la taille des données sources sont entièrement contrôlées par l'attaquant. L'adresse de destination est relative à un *offset*, lui aussi sous le contrôle d'un attaquant. Ce bug peut donc être transformé en une primitive d'*écriture relative* : l'attaquant a la capacité d'écrire des données de son choix, à un *offset* contrôlé d'un objet alloué dans le tas.

Les mécanismes de gestion du tas sont standards et ne sont pas durcis, sa structure est donc prédictible. Il est possible d'effectuer une suite d'appels déterministe aux fonctions PKCS #11 afin de mettre le tas dans un certain état où le déclenchement de la vulnérabilité provoque l'écrasement de la table de pointeurs d'un objet PKCS #11. L'exécution du programme peut alors être redirigée vers un *shellcode* présent sur le tas, qui s'avère être exécutable.

Par souci de brièveté, quelques détails techniques sont omis dans la description de l'exploitation. La taille des données copiées lors de l'écriture relative est par exemple limitée, et l'architecture PowerPC possède des caches différents pour les données et les instructions ; quelques astuces sont donc nécessaires pour obtenir un exploit stable. L'absence de mitigation, et notamment d'ASLR, rend le développement d'un exploit plus facile. L'utilisation du débogueur sur le HSM se révèle aussi d'une grande aide pour déterminer l'état de la mémoire pendant les différentes étapes de l'exploitation.

Impact Une fois l'exécution de code arbitraire obtenue se pose la question du choix du payload. Il est exécuté avec les droits *root*. Toutes les actions sont donc envisageables : récupération des clés, lecture de la mémoire pour récupérer des secrets, dump de la flash. . .

La solution retenue est un *patch* de la fonction de vérification du PIN, pour nous connecter en tant qu'administrateurs avec n'importe quel mot de passe. L'administrateur a les droits d'installer des modules. La charge suivante est un module récupérant l'intégralité de la flash chiffrée, et sa clé de déchiffrement. Le contenu est ensuite déchiffré hors connexion, révélant l'ensemble des secrets contenus dans le HSM.

L'exploit est un simple binaire à exécuter sur l'hôte.

3.7 Contournement de la signature du firmware

Le firmware et les modules sont signés. La procédure pour installer ou mettre à jour un firmware ou un module est similaire : il s'agit dans les deux cas d'une utilisation légèrement détournée d'un des mécanismes PKCS #11, à savoir la vérification de signature.

L'installation d'un firmware ou d'un module se fait avec un outil de configuration fourni avec le SDK. Cet outil communique uniquement avec l'interface *Cryptoki* du HSM. Nous expliquons tout d'abord le processus d'installation d'un module. Nous détaillerons ensuite la différence avec l'installation d'un firmware.

Installation d'un module Tout d'abord, l'administrateur ajoute un certificat avec une clé publique RSA sur le HSM. Ce certificat servira à vérifier la signature du module transmis par l'outil de configuration. Il doit être marqué comme un certificat de confiance ; cela requiert l'ajout de l'attribut `CKA_TRUSTED`. Cet attribut peut être rajouté par l'officier de sécurité. Une fois cette étape terminée, le module peut être envoyé au HSM.

L'outil effectue une opération de signature avec le mécanisme propriétaire `CKM_MOD_DOWNLOAD` ou `CKM_MOD_DOWNLOAD_2`. Les fonctions `C_VerifyInit`, `C_VerifyUpdate` et `C_VerifyFinal` vont vérifier que les données envoyées, c'est-à-dire l'intégralité du fichier du module, sont correctement signées. Le *handle* du certificat passé à `C_VerifyInit` est celui du certificat précédemment ajouté. Il doit avoir l'attribut `CKA_TRUSTED`. Les données passées à `C_VerifyUpdate` sont les données du module. La signature est générée lors de la compilation du module. Il s'agit d'une signature RSA PKCS #11 avec SHA-1 (`CKM_MOD_DOWNLOAD`) ou SHA-512 (`CKM_MOD_DOWNLOAD_2`).

Si la signature est correcte, le module est écrit sur la flash et le HSM est redémarré. Le module peut alors être utilisé.

Installation d'un firmware Le mécanisme d'envoi d'un firmware est très similaire, mais plus restrictif : le certificat vérifiant le firmware à flasher, signé par le constructeur, est fixe. Il est stocké dans le code du firmware et ne peut être modifié. L'outil de configuration ne connaît pas sa valeur ; il cherche parmi les objets du HSM un objet de type certificat avec un attribut propriétaire (0x80001337) devant valoir `true`. Cet attribut est en lecture seule, et ne peut jamais être ajouté à un certificat utilisateur. L'outil récupère le *handle* sur ce certificat et procède de la même manière que précédemment, cette fois avec les mécanismes `CKM_UPGRADE_SYS` et `CKM_UPGRADE_SYS_2`. Le HSM redémarre et le nouveau firmware est alors chargé.

Contournement de la signature du firmware Un problème de logique est présent dans ce code : l'attribut propriétaire 0x80001337 n'est pas vérifié lors de la vérification de la signature du firmware. Un administrateur peut donc signer n'importe quel firmware avec une clé qu'il aura lui-même générée. Si un certificat de confiance avec une clé publique correspondant à la clé privée est installé sur le HSM, le firmware sera considéré comme valide. Il suffit à l'administrateur de spécifier le *handle* de son certificat au lieu de celui du constructeur.

4 Impact et remédiation

4.1 Conséquences

D'utilisateur non authentifié à SO Les fonctions `C_GetOperationState` et `C_SetOperationState` peuvent être appelées par des utilisateurs non authentifiés, c'est à dire sans aucune connaissance du HSM ni des PIN des utilisateurs. L'exploitation de la confusion de type détaillée précédemment permet à un attaquant d'exécuter un code arbitraire sur le HSM, et donc d'obtenir les privilèges associés au rôle de l'administrateur. Cette vulnérabilité est exploitable depuis l'hôte, sans privilèges particuliers.

De SO à une compromission persistante Une fois le rôle administrateur obtenu, il est possible de mettre à jour le HSM avec un firmware signé par le constructeur. La vulnérabilité permettant de contourner la vérification de la signature du firmware autorise un attaquant à le remplacer par un firmware malveillant.

Cette vulnérabilité est vraiment problématique, car elle casse le mécanisme d'intégrité du HSM. Un utilisateur de HSM ne peut pas avoir de garanties quant à l'authenticité du firmware installé dans le HSM. Le code exécuté sur le HSM ne peut plus être considéré comme un code de confiance. Il devient possible d'installer un firmware incluant une *backdoor* sur le HSM. Plus grave encore, cette *backdoor* peut être persistante et survivre à d'autres mises à jour. L'éditeur a corrigé ce problème en vérifiant que le certificat présenté lors de la mise à jour possédait bien l'attribut 0x80001337. Cela ne résout que partiellement le problème : une version antérieure, vulnérable du HSM peut toujours être restaurée. Un attaquant pourra effectuer un *downgrade* du firmware, puis installer un nouveau firmware qu'il contrôle.

4.2 Durcissement logiciel

Plusieurs mécanismes peuvent être mis en oeuvre par l'utilisateur du HSM indépendamment du durcissement du firmware lui-même.

Chiffrement du firmware Les modules sont uniquement signés, comme le firmware. Or, nous ne souhaitons pas déployer des modules non chiffrés sur les HSM entreposés dans des *data centers*.

Le mécanisme de mise à jour des firmwares a donc dû être modifié afin d'ajouter la possibilité de charger des firmwares chiffrés et signés.

— Le chiffrement du firmware offre la possibilité de stocker des secrets sans risquer leur récupération par un attaquant.
— La signature des mises à jour assure que seules des mises à jour officielles peuvent être installées.

Désactivation partielle de PKCS #11 Le SDK utilisé pour développer les modules permet de modifier la table des pointeurs de fonctions PKCS #11. Il devient alors possible de réduire drastiquement la surface d'attaque en redirigeant les fonctions vers des filtres assurant la validité des arguments, ou ajoutant une authentification supplémentaire.

Des vulnérabilités peuvent cependant toujours être présentes dans l'implémentation des algorithmes de décodage des objets PKCS #11, appelées avant leur passage aux fonctions PKCS #11. Quelques fonctions ne peuvent pas être filtrées ; il est particulièrement important de s'assurer que celles-ci ne contiennent pas de vulnérabilités. Une fois le filtrage mis en place, nos efforts se sont concentrés sur ces fonctions.

Ce mécanisme de *hook*, s'il est correctement implémenté, empêche le déclenchement de la plupart des vulnérabilités découvertes durant l'étude.

Discussions avec le fabricant Nous remercions l'équipe sécurité du constructeur avec qui nous avons toujours eu des échanges constructifs. Toutes les vulnérabilités rapportées ont été corrigées rapidement. Les versions bêta des firmwares ont été mises à notre disposition ; nous les avons analysées pour nous assurer que les correctifs étaient suffisants, ce qui a toujours été le cas.

Il serait intéressant d'ajouter des mitigations supplémentaires sur cette plateforme. En effet, l'ajout d'ASLR, d'une réelle isolation des modules, d'un *stack cookie*, d'un tas non exécutable, et des options de durcissement du noyau à la compilation, rendraient la tâche bien plus difficile à un attaquant qui trouverait une vulnérabilité dans les interfaces proposées. Ce sera probablement l'objet d'une mise à jour future.

5 Conclusion

Il est difficile de différencier le niveau de sécurité des HSM disponibles sur le marché, et les certificats délivrés ne sont pas forcément représentatifs des attentes des utilisateurs. L'intérêt de l'analyse sécuritaire des produits est double, aussi bien pour confirmer que le niveau de sécurité correspond à celui attendu que pour avoir une meilleure connaissance du produit et mieux comprendre les risques qu'il présente.

Cette étude n'est cependant pas exhaustive et n'a pas la prétention d'être un état de l'art de la sécurité de l'ensemble des modèles de HSM disponibles sur le marché. Il est fort possible que le niveau de sécurité varie fortement d'un constructeur à l'autre, et même entre différents modèles d'un même constructeur.

Concernant le modèle analysé, il est regrettable qu'aucun durcissement ne soit appliqué au firmware d'un produit de sécurité. En effet, même s'il est impossible de garantir l'absence de vulnérabilités, complexifier le travail d'un attaquant devrait être une priorité. Enfin, les vulnérabilités présentées sont exploitées sur la version PCI du modèle de HSM étudié. Il serait intéressant de déterminer si les versions réseau de cette gamme sont aussi vulnérables, et exploitables à distance sans accès à la machine hôte.

Références

1. Jean-Philippe Aumasson, Yolan Romailler. Automated Testing of Crypto Software Using Differential Fuzzing. Black Hat USA, 2017.
2. Ryad Benadjila, Thomas Calderon, and Marion Daubignard. Buy it, use it, break it... fix it : Caml Crush, un proxy PKCS#11 filtrant. *SSTIC*, 2014.
3. Daniel Bleichenbacher, Thai Duong, Emilia Kasper, and Quan Nguyen. Project Wycheproof. `https://github.com/google/wycheproof/`.
4. Jolyon Clulow. On the Security of PKCS #11. In Colin D. Walter, Çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003*, pages 411–425, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
5. Matus Nemec, Marek Sys, Petr Svenda, Dusan Klinec, and Vashek Matyas. The Return of Coppersmith's Attack : Practical Factorization of Widely Used RSA Moduli. In *24th ACM Conference on Computer and Communications Security (CCS'2017)*, pages 1631–1648. ACM, 2017.
6. OASIS Open. PKCS #11 Cryptographic Token Interface Base Specification Version 2.40, 14 avril 2015.
7. Cem Paya. Your Bitcoin Wallet May Be At Risk : Safenet HSM Key-Extraction Vulnerability. `https://medium.com/gemini/your-bitcoin-wallet-may-be-at-risk-safenet-hsm-key-extraction-vulnerability-58c97bf6b927`, 10 juin 2015.
8. Cryptosense SA. HSM Security. Securing PKCS#11 Interfaces. Technical report, 2018.
9. Graham Steel, Riccardo Focardi, Matteo Bortolozzo, and Matteo Centenaro. Attacking and Fixing PKCS#11 Security Tokens with Tookan. SSTIC, 2011.
10. Utimaco. CVE-2015-6924 Elliptic Curve key disclosure vulnerability. `https://support.hsm.utimaco.com/documents/20182/39856/CVE-2015-6924+Security+Advisory`, 2015.