Compromission d'un terminal sécurisé via l'interface carte à puce

Guillaume Vinet
g.vinet@serma.com

SERMA Technologies, CESTI, 30, avenue Gustave Eiffel
33600 Pessac, France

Résumé De nombreuses applications sur PC utilisent une carte à puce pour renforcer leur sécurité, par exemple dans le domaine de la signature numérique. C'est un excellent coffre-fort numérique à faible coût, mais dont la sécurité repose très souvent sur l'envoi d'un code fixe en clair : le code PIN. Il est donc impératif de le saisir dans un environnement sécurisé, et parfois hasardeux de le faire sur un PC. De petits terminaux à connexion USB disposant d'un écran et d'un clavier proposent une solution en cloisonnant la manipulation du PIN en leur sein. Cet article s'intéresse aux attaques sur ces terminaux via l'interface carte à puce. Leur fonctionnement nécessite des attaques au niveau du protocole de communication bas niveau. Cela a nécessité la création de notre propre émulateur de carte à puce. De plus, leur architecture rudimentaire diminue les moyens d'exploitation de failles éventuelles. Malgré ces limitations, la découverte de plusieurs failles associées à des exploitations concrètes démontre la pertinence de cette approche.

1 Introduction

La dématérialisation des supports d'information est de plus en plus importante. Pourtant, des obstacles s'opposent à son déploiement comme la nécessité de garantir la non-répudiation ou l'intégrité d'un document signé. Certaines cartes à puces telles que la future carte d'identité électronique proposent des solutions en fournissant un mécanisme de signature numérique. Équipé d'un PC, d'un lecteur de carte à puce et de l'application adéquate, n'importe quel utilisateur peut désormais signer ses documents en entreprise ou à domicile. Néanmoins, pour être accessible, cette opération requiert généralement la saisie d'un bon code PIN pour vérifier l'identité du possesseur de la carte. La sécurité de sa manipulation, tant par la carte à puce que par le lecteur, est donc cruciale.

La carte à puce est reconnue comme un excellent coffre-fort numérique grâce à ses exigences matérielles et logicielles très élevées. Le stockage, l'accès ou l'utilisation des données secrètes sont sûrs. Cependant

dans la plupart des cas, l'utilisation de ce coffre-fort est uniquement soumis à la bonne validation d'une donnée numérique fixe : le code PIN. Sa saisie est le principal point faible d'une carte à puce car elle s'accomplit nécessairement en clair. Cette opération doit donc être effectuée dans un environnement sécurisé.

Dans une utilisation classique, la saisie du code PIN s'effectue sur le clavier du PC interfacé à la carte via un lecteur. Mais comme le PC n'est pas réputé comme étant l'environnement le plus sûr, cette utilisation est très risquée [16]. Une sécurisation du PC est toujours possible, mais elle est souvent coûteuse et contraignante.

Afin que ce problème ne soit pas un frein au déploiement de ces solutions dans les administrations ou chez les particuliers, les industriels ont développé une solution aujourd'hui répandue. Elle consiste en un petit terminal équipé d'un écran et d'un clavier. Il s'interface avec une carte à puce, et possède une connexion USB pour se brancher au PC. Ce terminal propose un mode où la saisie du code PIN s'effectue grâce à son clavier. Ainsi, la récupération, la manipulation et l'envoi du code PIN sont confinés au sein du terminal. Néanmoins, cette solution ne fait que transférer le problème de sécurité d'un système à un autre. La question reste toujours la même : la manipulation du code PIN, sur cet environnement, est-elle sécurisée ?

Le périmètre et la complexité de ces terminaux sont beaucoup plus restreints que pour un PC. Bien que plus simples à analyser, ils restent soumis à de nombreuses façons de les compromettre. Cet article va s'attacher à analyser la sécurité de ce type de produits à partir de l'interface de la carte à puce.

Notre article débutera d'abord par l'analyse du fonctionnement de ce type de terminal. Nous verrons que ces terminaux sont totalement transparents par rapport aux échanges applicatifs de la carte. Les attaques ne peuvent être menées qu'au niveau du protocole de communication ISO/IEC 7816-3 [7], sur lequel nous effectuerons une analyse de vulnérabilité. Suite à cela, nous ferons un retour d'expérience sur la mise en pratique de ces attaques. Nous détaillerons l'architecture rudimentaire des terminaux visés, ainsi que la mise en place de nos tests par fuzzing qui a nécessité la création d'un nouvel outil, un émulateur de carte à puce. Finalement, nous détaillerons la découverte et l'exploitation de plusieurs failles logicielles qui démontrent la pertinence de cette approche.

2 Un simple lecteur de carte à puce

En apparence, ces terminaux ressemblent très fortement aux terminaux bancaires que l'on trouve, par exemple, chez les commerçants. Ils possèdent, en effet, un affichage numérique sur deux lignes, un clavier numérique, et des touches de validation, d'annulation ou de correction.

2.1 Un lecteur de carte à puce USB classique

En faisant abstraction de l'écran et du clavier, nous nous retrouvons face à un classique lecteur de carte à puce se branchant en USB sur le PC (figure 1).

La communication entre le terminal et la carte puce suit la spécification ISO 7816-3. Deux structures constituées chacune de caractères hexadécimaux entrent en jeux. La première d'entre elles est l'APDU (Application Protocol Data Unit) qui est directement manipulée par l'application de la carte. Pour être envoyée au terminal, elle doit subir plusieurs transformations. La nouvelle structure obtenue est alors une TPDU (Transport Protocol Data Unit).

Les échanges entre le terminal et le PC suivent la spécification CCID (Circuit(s) Cards Interface Device [4]) qui définit un jeu de commandes USB pour détecter par exemple l'insertion/l'extraction d'une carte, pouvoir la mettre sous tension ou bien envoyer/recevoir des commandes. De nos jours, elle est implémentée par la plupart des lecteurs.

Finalement pour le dialogue entre une application sur le PC et le terminal, le moyen le plus simple est d'utiliser la norme PC/SC (Personal Computer/Smart Card) [19]. Elle définit une API permettant d'accéder de manière unifiée à un lecteur, et donc à la carte, quel que soit le type de connexion (USB, série...), tout en gardant des communications au niveau APDU. Une implémentation est disponible sous Windows, ainsi que sous GNU/Linux et MAC OS X (PC/SC Lite [15]).

Dans la plupart des applications sur carte à puce (bancaires [9,18] ou de signature électronique [6]), l'authentification du porteur de la carte s'effectue grâce à un code numérique nommé code PIN. Sa validation s'effectue en l'envoyant en clair dans une commande APDU appelée « Verify PIN » qui est définie dans la norme ISO/IEC 7816-4 [8] (voir la table 1 pour un exemple avec le code PIN 1234 au format EMV). Sa validation autorise par exemple la lecture de données protégées ou l'accès à des mécanismes de signature utilisant une clé secrète. Tous ces services, dont le bon fonctionnement nécessite la validation du code PIN, seront

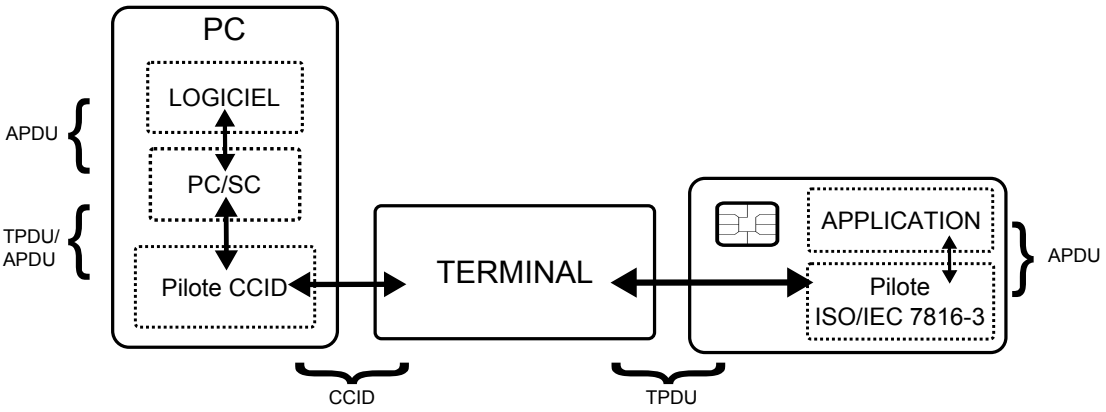

Figure 1. Fonctionnement en tant que lecteur de carte à puce.

appelés « services de sécurité ». Certaines applications supportent l'envoi d'un PIN chiffré. Cette alternative n'est pas souvent utilisée car elle nécessite la mise en place d'une infrastructure cryptographique lourde et soulève d'autres problèmes comme l'accès à la clé secrète de chiffrement. De plus, le PIN doit encore être saisi en clair avant d'être chiffré, puis envoyé.

00 20 00 80 08 24 12 34 FF FF FF FF FF 00

Table 1. Commande Verify PIN.

Dans le cas décrit dans la figure 1, nous avons une application installée sur le PC et connectée, grâce à un lecteur, à une carte à puce. Pour tirer partie des services de sécurité de cette carte, elle doit au préalable lui transmettre le bon code PIN. L'application requiert sa saisie sur le PC via le clavier ou la souris (dans le cadre d'un clavier virtuel) puis l'envoie à la carte. Si le PC est contaminé, par exemple par un logiciel espion, le code PIN peut être intercepté.

2.2 Un terminal de saisie sécurisé

Pour pallier ce problème, ces terminaux possèdent un deuxième mode de fonctionnement qui tire partie de leur clavier et de leur écran. Il est défini dans la partie 10 de la spécification PC/SC : « IFDs with Secure PIN Entry Capabilities » [20] et est pris en compte dans la norme CCID.

Dans celui-ci, le PC envoie une requête au terminal en lui demandant de vérifier le code PIN par lui-même. Le terminal se charge alors de le demander avec son écran et de le récupèrer via son clavier. Il construit ensuite la commande « Verify PIN » avec le PIN saisi, puis l'envoi à la carte. Finalement, il transmet la réponse à l'ordinateur.

La spécification ne propose que l'envoi du code PIN en clair. Par commodité, nous appellerons ce mode « Secure Verify PIN » (voir figure 2). Une deuxième commande manipulant le code PIN fonctionne aussi sur le même principe. Il s'agit de la commande « Modify PIN » (définie dans la norme ISO/IEC 7816-4 [8]). Elle permet de remplacer le code PIN de la carte, et contient donc à la fois l'ancienne et la nouvelle valeur en clair. Nous appellerons cette version « Secure Modify PIN ».

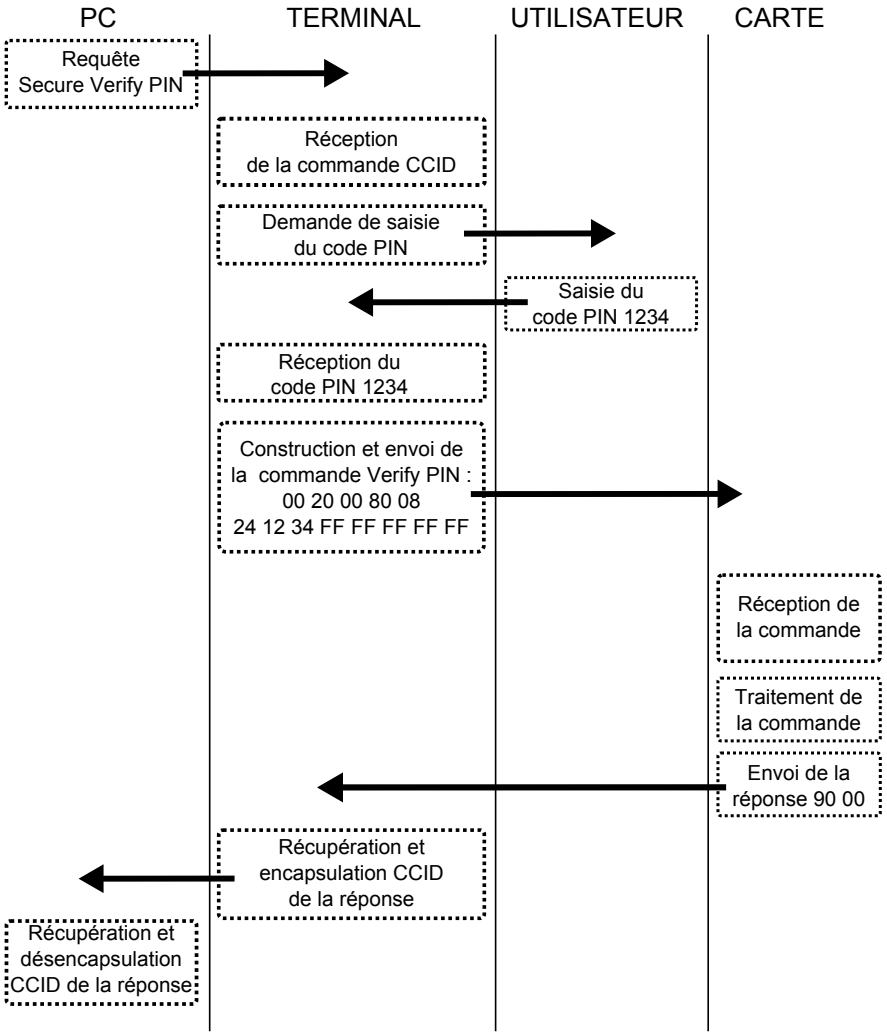

Figure 2. Fonctionnement du Secure Verify PIN.

2.3 Modèle de sécurité

Le but de ce terminal est donc de protéger le code PIN en le cloisonnant en son sein (hormis son envoi à la carte) : le PC ne le manipule jamais. En effet au cours de cette transaction, la requête envoyée par le PC ne contient que des options de configuration : structure de l'APDU, délai d'attente, langue du message affiché ... Tandis que la réponse du terminal n'indique que le succès ou l'échec de la transaction. La réponse du terminal récupérée par le PC peut être modifiée par un logiciel malveillant pour faire croire à l'application sur le PC que la vérification a réussi. Cela reste cependant inutile puisque l'accès aux services de sécurité sous-jacents n'est toujours pas accordé : les étapes suivantes échoueront.

Cependant, l'interrogation de la carte à puce en lui envoyant des commandes via le terminal n'est pas protégée. Ainsi une fois le PIN vérifié, un logiciel espion installé sur le PC peut légitiment accéder aux services de sécurité activés, et donc par exemple aux fichiers protégés le temps d'une session. Cette problématique n'entre pas dans le modèle de sécurité de ce terminal. Son traitement est laissé aux applications sur la carte. Un des moyens existants est la protection des échanges en instaurant un *Secure Messaging*. Une clé éphémère est échangée entre les deux parties (en se basant sur le protocole Diffie-Hellman). Puis, chaque message échangé est protégé en confidentialité (chiffrement des données) et en intégrité (calcul d'un MAC).

Certains terminaux implémentent cette spécification tout en s'enrichissant d'autres fonctionnalités : connexion ethernet, imprimante à ticket, vérification de la transaction bancaire ... Nous nous focaliserons sur les terminaux de base, c'est à dire ceux implémentant simplement cette norme [20].

3 Attaques via l'interface carte à puce

L'attaque de ces terminaux consiste à récupérer un code PIN saisie via une des deux commandes sécurisées lors d'une précédente session. Au passage, nous pouvons aussi essayer de récupérer le maximum de données des transactions précédentes.

De nombreux moyens sont envisageables pour réussir ces opérations. Avant la saisie du code PIN, le terminal peut être piégé en modifiant physiquement son micro-contrôleur, ou bien en installant un espion matériel sur le port d'insertion de la carte à puce. Pendant la manipulation du terminal par une personne légitime, un attaquant peut espionner les rayonnements électromagnétiques du lecteur pour déterminer à distance

les touches frappées, et donc récupérer le code PIN. Enfin après sa saisie, le terminal peut encore être démonté pour analyser si le PIN n'est pas présent dans une mémoire de masse (EEPROM, FLASH) ou temporaire (RAM). Des attaques logicielles peuvent aussi être mise en oeuvre. Elles peuvent se pratiquer au niveau du clavier, de l'interface USB ou bien de la carte à puce. Cet article se concentrera sur cette dernière interface. Les attaques via l'interface de la carte à puce sont encore relativement rares [11]. Elles sont néanmoins intéressantes puisqu'elles sont rapidement mise en oeuvre, non destructives, et donc difficilement détectables.

La mise en place de ces attaques va donc nécessiter d'avoir un accès à un terminal ayant précédemment servi à saisir avec succès le code PIN d'un utilisateur. Une carte malveillante renvoyant des données incorrectes est utilisée pour l'attaquer, mais dans la majeure partie des cas elle est esclave du terminal. Il faut pouvoir stimuler son action malveillante en envoyant des commandes USB au terminal. Pour le faire, l'attaquant peut directement utiliser le PC auquel est relié le terminal. Dans le cas où il n'est pas accessible (son écran est verrouillé par exemple), il peut débrancher le terminal tout en le gardant sous tension grâce au dispositif adéquat (« *warm replug attack* » [10]) dans le but d'éviter l'effacement de sa mémoire volatile. L'attaquant pourra alors le rebrancher sur son propre PC de test pour l'analyser tranquillement. Une fois le PIN récupéré et la carte associée volée, il est donc possible de l'utiliser à sa guise. Dans le cas d'une carte bancaire, l'attaquant pourra retirer l'argent qu'il désire à l'insu de son véritable propriétaire.

4 Analyse de vulnérabilité du protocole de communication ISO 7816-3

Qu'il soit vu comme un lecteur de carte à puce ou comme un moyen de saisie sécurisé du code PIN, ce terminal demeure toujours totalement transparent par rapport aux applications embarquées sur la carte. En effet, il n'a pas pour but d'analyser les réponses de la carte, par exemple en décodant et extrayant des champs ou en effectuant des calculs cryptographiques. Il ne fait que les transmettre au PC, qui lui va les interpréter. Cette caractéristique lui assure une compatibilité avec la majeure partie des applications sur carte à puce puisque l'authentification du porteur par code PIN est très répandue.

De notre coté, cela signifie que la couche applicative ne peut pas être attaquée. Nous devons nous focaliser sur la couche inférieure définissant le protocole de communication bas niveau : la norme ISO/IEC 7816-3 [7].

Cette approche est contraignante puisqu'elle réduit fortement la surface d'attaque. Cependant, son intérêt est de pouvoir s'appliquer à n'importe quel type de terminal communiquant en contact avec une carte à puce, quelle que soit l'application installée dessus.

L'ancienneté de cette norme lui assure normalement une implémentation bien maîtrisée. Mais, elle reste encore complexe sur certains points. Notre analyse de vulnérabilité va consister à détailler les différentes étapes de la communication, puis lister pour chacune d'entre elles, les structures reçues par le terminal. Nous déterminerons les champs dont une valeur erronée permettrait d'obtenir un comportement anormal suite à une implémentation incorrecte ou trop laxiste de la spécification. Plus particulièrement, nous nous focaliserons sur les valeurs induisant en erreur le terminal sur la taille des données reçues, et dont une conséquence possible serait le déclenchement d'un débordement de tampon.

4.1 L'Answer To Reset (ATR)

L'ATR est envoyé par la carte dès qu'elle est correctement mise sous tension par le terminal. Il donne plusieurs informations au terminal, ce qui lui permet de connaître ou de personnaliser certains paramètres de la communication. N'étant pas de taille fixe, il possède une structure particulière qui permet au lecteur de calculer combien d'octets il va recevoir, et doit donc lire.

Un ATR est constitué au minimum de deux octets. Le premier octet, noté TS, indique le codage des caractères sur la ligne physique à utiliser pour la suite de la communication. Ensuite, chaque quartet du deuxième octet, noté T0, donne des informations différentes.

Le quartet de poids faible indique la taille d'un champ variable situé à la fin de l'ATR, appelé « octets historiques ». Ce champ peut donc contenir de 0 à 15 octets. Ce champ optionnel peut parfois contenir des renseignements sur les capacités de la carte. (voir la table 2).

ATR	3B 03 45 78 68
3B	codage des octets en convention directe
03	la suite de l'ATR contient 3 octets historiques
45 78 68	octets historiques

Table 2. ATR avec octets historiques.

Dans le quartet de poids fort de l'octet T0, chaque bit indique la présence ou non d'octets de configuration, notés de TA1, TB1, TC1, TD1 (voir la table 3). Le quartet de poids fort de l'octet TD1 fonctionne sur le même principe : il indique la présence d'octets de configuration TA2 à TD2, et ainsi de suite. Pour éviter une récurrence infinie, la norme impose que la taille de l'ATR soit inférieure à 33 octets. Tous ces octets de configuration ont un impact sur la communication qui va être instaurée :

- protocole de transport disponibles : T0, T1,
- vitesses de communication supportées par la carte,
- les paramétrages disponibles sur les temps d'attentes,
- ...

ATR	3B 13 96 45 78 68
3B	codage des octets en convention directe
13	L'ATR contient 3 octets historiques et l'octet de configuration TA1
96	TA1
45 78 68	octets historiques

Table 3. ATR avec octet de configuration.

Des valeurs erronées dans les octets T0 et TD*X* permettraient donc d'induire en erreur le terminal sur la taille réelle de l'ATR reçu.

4.2 APDU et TPDU

Une fois l'ATR reçu, le terminal peut choisir entre deux modes de communication suivant les possibilités de la carte : le mode T0 ou le mode T1. Ce choix s'effectue lors de la négociation PPS (« Protocol and Parameters Selection »), cet échange n'entraînant pas de vulnérabilité particulière, nous ne le détaillerons pas.

Ces modes n'ont pas d'impact sur le codage d'un caractère sur la ligne physique, mais sur l'envoi de la structure de base de la communication, l'APDU, qui peut être de deux types :

- la commande APDU qui va du terminal vers la carte (voir la table 4). Elle est constituée :
  - d'un entête de 5 à 6 octets notés : « CLA, INS, P1, P2, P3 » (le champ P3 pouvant varier de 1 à 2 octets),
  - d'un corps de données de « P3 » octets,
  - d'un champ « Le ». Il indique à la carte le nombre d'octets de réponse attendu par le terminal. Il est facultatif.

Entête	Corps de données	Le
00 20 00 80 08	24 12 34 FF FF FF FF FF	00

Table 4. Commande APDU.

– la réponse APDU qui va de la carte vers le terminal (voir la table 5). Elle est constituée :
  – d'un corps de données, de taille variable.
  – d'un champ de deux octets appelés « status words » (SW). Il indique la réussite de la commande, et le cas échéant, les raisons de son échec.

Corps de données	SW
87 48 64 78 45	9000

Table 5. Réponse APDU.

L'envoi d'une APDU est caractérisé par deux étapes. Tout d'abord comme indiqué auparavant, l'APDU peut subir des transformations : ajout ou suppression d'octets, encapsulation. La nouvelle structure obtenue est nommée TPDU (Transmission Protocol Data Unit). Finalement, c'est cette TPDU qui est envoyée et acquittée entre le terminal et la carte. Le choix du mode (T0 ou T1) a un impact à la fois sur la construction de la TPDU, et ainsi que la manière de l'envoyer.

Les octets de procédures du mode T0

Le mode T0 se fonde sur une communication par caractères. L'APDU ne subit quasiment aucune modification lors de sa transformation en TPDU. En contrepartie, son envoi peut être extrêmement segmenté. Il nécessite beaucoup plus d'échanges et d'acquittements entre les deux entités pour savoir si le caractère reçu est effectivement le dernier de la transaction.

Lors de l'envoi d'une commande APDU, le terminal commence d'abord par envoyer les cinq premiers octets de l'entête. Ensuite, il attend la réception d'un caractère particulier envoyé par la carte, appelé octet de procédure qui démarre le dialogue entre la carte et le terminal (voir l'exemple de la figure 3) :

- Un octet « NULL » de valeur 0x60. Il indique au terminal de ne pas faire d'action, et d'attendre la réception d'un autre octet de procédure.
- Un octet « SW1 », premier octet de « status word ». Il vaut 0x6X ou 0x9X. Le terminal attend alors le deuxième octet du « status word », SW2. L'envoi de la commande est terminée. Le terminal se met en attente d'une éventuelle réponse de la carte.
- Un octet d'acquittement « ACK ». Il peut être de deux sortes.
  - Si sa valeur correspond au deuxième octet de l'entête, noté INS, le terminal peut envoyer le reste des octets de la commande. Dès qu'il a fini, il attend un nouvel octet de procédure de la part de la carte.
  - Si sa valeur correspond à INS xor 0xFF, le terminal ne peut envoyer qu'un octet du reste de la commande. Il attend ensuite un nouvel octet de procédure.

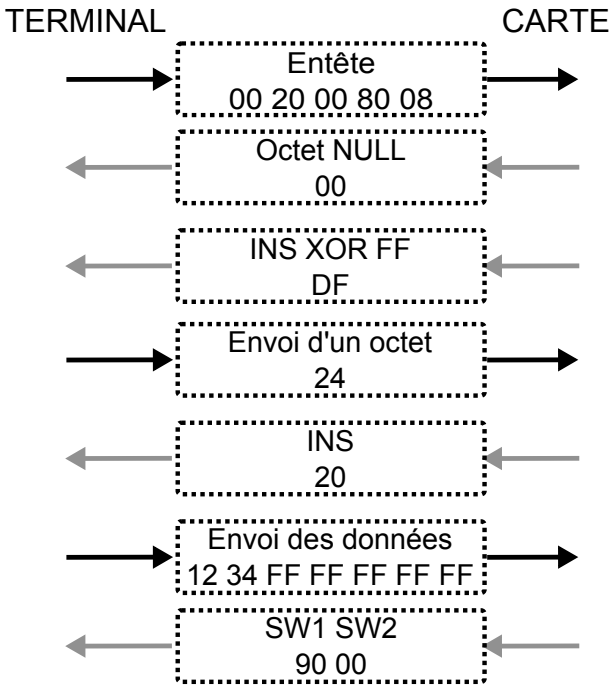

Figure 3. Exemple d'envoi de la commande « Verify PIN » en mode T0.

Cette machine à état complexe est propice à une mauvaise implémentation, plus particulièrement sur l'envoi des octets de procédures par exemple avec la réception octet par octet de la commande du terminal ou l'envoi d'un grand nombre d'octets « NULL ».

Les blocs de données du mode T1

Contrairement au mode précédent, le mode T1 se fonde sur une communication par blocs. L'APDU est transformée en TPDU en l'encapsulant dans une structure constituée d'un prologue et d'un épilogue. L'indication de la taille du bloc dans le prologue permet un envoi quasi direct puisque le destinataire peut connaître tout de suite la taille du message qu'il reçoit.

Le prologue est constitué de 3 champs d'un octet chacun :

- Le Node Address Byte (NAD). Il permet d'identifier la source et le destinataire du bloc. La plupart du temps, ce champ n'est pas utilisé et est positionné à la valeur 0x00.
- Le Protocol Control Byte (PCB). Il indique le type de bloc envoyé. En effet, un bloc peut contenir une APDU, ou bien des informations de contrôle.
- Le champ de longueur renseignant la taille de l'APDU envoyée.

L'épilogue est constitué d'un octet de CRC (« *Cyclic Redundancy Check* ») ou de deux octets de LRC (« *Longitudinal Redundancy Check* ») suivant la configuration de la communication. Il permet de vérifier l'intégrité du bloc reçu.

Prologue	APDU	Epilogue
00 00 0C	00 A4 04 0C 07 A0 00 00 02 47 10 01	53
00 00 02	90 00	92

Table 6. Exemple d'une commande/réponse TPDU en mode T1.

En plus d'essayer de mettre des valeurs incorrectes, la principale voie d'attaque sera de mettre un champ de longueur incohérent avec la taille réelle de l'APDU.

Un autre chemin d'attaque peut être employé. Comme indiqué précédemment, plusieurs types de blocs existent. L'un des plus intéressant est celui permettant de modifier la taille du bloc (champ de longueur), qui peut alors être limitée à 40 octets par exemple. Quand cette taille est inférieure à celle de l'APDU, cette dernière est segmentée en plusieurs partie et autant de blocs de TPDUs seront alors envoyées et acquittées. Une réponse trop grande de la carte est normalement rejetée. La modification de la taille du bloc peut intervenir à tout moment et plusieurs fois

lors d'une communication (voir table 7). Cette segmentation peut être propice à une mauvaise implémentation de la part du lecteur.

Prologue	Champ de données	Epilogue
00 C1 01	FE	3E
00 E1 01	FE	1E

Table 7. Echange positionnant la taille de bloc à 0xFE octets.

4.3 Les APDU étendues

La taille du corps de données d'une APDU peut être codée sur un ou deux octets. Sa taille maximale peut donc être soit de 256 ou de 65536 octets (à chaque fois la valeur nulle code la valeur maximale). Dans ce dernier cas, on parle alors d'APDU étendue.

De plus en plus d'application sur carte à puce (et donc de terminaux) supportent ce mode, par exemple pour la signature numérique. Néanmoins, la taille de la mémoire d'une carte à puce (et de certains lecteurs) est encore très limitée de nos jours. Ainsi, si ce mode est implémenté, la taille maximale supportée est nettement limitée, par exemple 1000 octets. Un terminal trop laxiste dans sa gestion de la mémoire et de la réception des fragments de commande APDU peut donc être vulnérable à un débordement de tampon.

5 Outillage

Maintenant que nous avons caractérisé l'interface à attaquer et réalisé l'analyse de vulnérabilité, il reste à mettre en œuvre concrètement ces attaques.

5.1 Emulateur de carte à puce

Pour réaliser nos attaques sur le protocole ISO 7816-3, les outils disponibles publiquement ne répondaient pas à nos exigences :

- Le premier moyen était l'achat dans le commerce de carte à puce programmable, par exemple des cartes proposant une plate-forme JavaCard. Néanmoins dans ce cas, la couche de transport bas-niveaux est déjà implémentée. Ainsi, seule l'attaque via les APDUs étendues pouvait être utilisée.

- D'autres types de carte programmable permettent une personnalisation totale de la couche de transport (Fun Card, Prussian Card). Le problème est qu'elles doivent être reprogrammées en dialoguant avec un lecteur de carte à puce. Or, le but de nos tests est justement de générer des communications incorrectes. C'est à dire que nous ne pourrons plus les reprogrammer puisqu'un lecteur nominale sera incapable de communiquer avec.
- Des émulateurs physiques de carte à puce existent. Ils ont un port de communication en plus de celui de la carte à puce (USB par exemple). Ils peuvent donc implémenter une communication ISO 7816-3 incorrecte, tout en pouvant être reprogrammés ultérieurement. Cependant, ils peuvent avoir encore plusieurs contraintes : leur coût, leur faible degré de personnalisation ou bien la difficulté à les automatiser.

Notre approche a donc consisté à créer notre propre émulateur à base d'Arduino [2]. Arduino est une carte électronique qui est très utilisée dans le domaine du matériel libre. C'est un micro-contrôleur (souvent basé sur une puce ATMEL AVR ou une puce ARM) qui peut être programmé pour analyser et produire des signaux électriques. Son très faible coût, sa puissance et son extrême facilité de prise en main en font un outil particulièrement adapté pour la réalisation de projets électroniques. Les versions les plus récentes possèdent une connexion USB pour permettre une connexion, une programmation et un pilotage via un PC. Un kit de développement très complet, libre et aussi utilisable sous l'environnement Eclipse est disponible. Arduino possède une communauté d'utilisateurs extrêmement active ce qui assure la disponibilité de nombreux outils, bibliothèques ou forums d'entraide. Depuis la réalisation de notre outils, de nombreuses alternatives à Arduino sont apparues : Pinguino [2], Raspberry [3] . . . Néanmoins, Arduino répondant encore à nos besoins, nous n'envisageons pas de migrer vers une autre plate-forme.

L'idée de base de notre émulateur est qu'il soit construit comme une sorte de passerelle entre un connecteur carte à puce et un PC (voir figure 4). Notre émulateur se compose donc de trois parties :

- Une « raquette » avec un connecteur ISO 7816-3. Elle est insérée dans le terminal.
- Cette « raquette » est ensuite reliée à notre Arduino au travers des connecteurs définis dans la norme ISO 7816-3 : Clock, Vcc, Reset, I/O, Ground. Nous utilisons l'Arduino comme passerelle. Il contient les briques de base pour générer les signaux électriques et envoyer des structures primaires.

– L'Arduino est relié via un câble USB au PC. Il est piloté grâce à un serveur python que nous avons développé. Celui-ci lui indique dynamiquement les commandes à envoyer en renseignant la suite de structure primaire à utiliser. Le moteur de modification du serveur peut opérer à plusieurs niveaux : l'APDU, la TPDU ainsi que l'envoi de la TPDU (octets de procédure par exemple).

Figure 4. Schéma de l'émulateur de carte à puce.

A partir de maintenant, nous considérons que les termes suivants désignent la même entité : émulateur de carte à puce, carte malveillante, émulateur, carte.

5.2 Tests par fuzzing protocolaire

Nous pratiquerons des tests de fuzzing pour détecter des vulnérabilités sur ces terminaux. Cette technique de tests en boîte noire consiste à envoyer des données générées de façon plus ou moins aléatoires, à les soumettre au produit, puis à détecter un comportement incorrect. Dans ce cas, une analyse plus poussée permet de déterminer si cela est le point d'entrée d'une éventuelle vulnérabilité.

Cet article n'a pas pour vocation de s'attarder sur son explication ou les outils pour les mettre en place : une littérature très abondante existe à ce sujet [13,17,5,1]. Nous allons plutôt expliquer rapidement le pilotage de nos attaques. Nous utilisons notre propre fuzzer basé sur le moteur de Sulley [12]. Sa programmation en python permet de dialoguer avec le serveur python de notre émulateur.

Le terminal à attaquer ainsi que notre émulateur sont tous les deux branchés en USB sur notre PC de test. La raquette de l'émulateur est insérée dans le terminal. Dans un premier temps, le fuzzer indique à l'émulateur la donnée de fuzzing à envoyer (via le serveur python). Cependant, l'émulateur (en tant que carte à puce) est esclave dans la communication. Donc dans un deuxième temps, le fuzzer doit piloter le lecteur en lui envoyant une commande valide qui va déclencher l'envoi

de notre réponse erronée (notre charge) par la carte. Nous récupérons ensuite la réponse du terminal pour déterminer l'impact de l'attaque (voir figure 5).

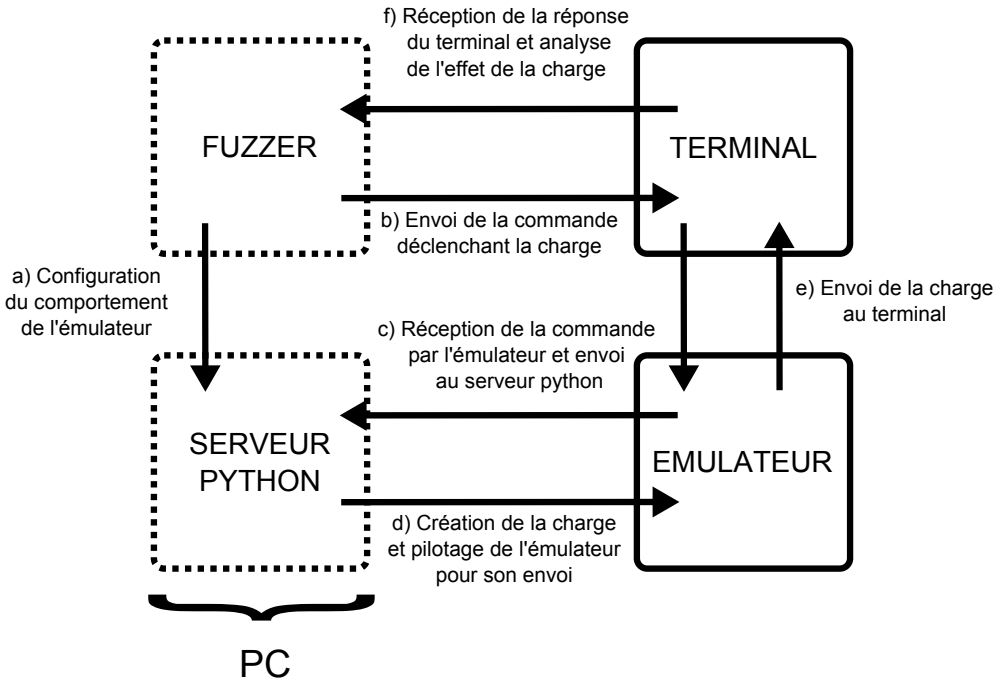

Figure 5. Pilotage de l'émulateur et du terminal par notre outil de fuzzing.

Un problème demeure néanmoins. En ne communiquant qu'au niveau APDU, ce qui est le cas par exemple lorsqu'on pilote le terminal avec PC/SC, un problème apparaît. En effet, des réponses pertinentes du terminal peuvent être perdues. La trame CCID renvoyée peut être mal formatée. Une trame CCID est constitué d'un enentête d'une taille fixe de 10 octets, et d'un champ optionnel de données. Le premier octet indique le type de la commande ainsi que son sens, par exemple du PC vers le lecteur. Les quatre octets suivant indiquent la taille du champ de données. Il est codé en *little-endian*. Les autres octets codent des informations tel que le numéro de séquence de la commande ou le numéro de slot. PC/SC vérifie la cohérence de ces paramètres, et ignore la commande si ils sont invalides. Il est donc important de travailler au niveau de la réponse CCID, tout en étant le plus laxiste possible au niveau des vérifications.

Sous Linux, le pilotage du lecteur se fait via une implémentation libre de PC/SC fournie avec des drivers CCID [15]. Comme le code source est disponible, il est très facile de le modifier pour récupérer les commandes CCID et supprimer le maximum de vérification sur le formatage. C'est

un bon moyen pour commencer, mais cette implémentation de PC/SC envoie souvent des commandes intempestives qui peuvent parasiter nos résultats. Cela devient plus compliqué de modifier les sources tout en gardant un résultat cohérent. Il est alors plus pratique de piloter directement le terminal en USB, par exemple avec la bibliothèque python pyUSB, et en réimplémentant la norme CCID.

6 Mise en oeuvre des attaques

Nous avons testé 5 terminaux à contact implémentant simplement la norme « IFDs with Secure PIN Entry Capabilities », c'est à dire de simples lecteurs de carte à puce avec une saisie sécurisée du PIN [1]. Ce choix a été poussé par deux raisons principales. La première, comme dit auparavant, est que ce type de terminal nous oblige à les attaquer au niveau du protocole de communication bas niveau ISO/IEC 7816-3. Nous voulions vérifier si cette surface d'attaque réduite pouvait être un vecteur d'attaque réaliste. La deuxième raison, que nous allons voir, est que ces terminaux « de base » ont un fonctionnement rudimentaire tout comme leur architecture interne. Ainsi, leur sécurisation devrait être triviale. Nous pouvions donc nous interroger sur la pertinence de les attaquer. En effet, peut-on vraiment s'attendre à trouver des failles sur ce type de terminaux, et si oui, réussir à les exploiter ?

6.1 Architecture des terminaux attaqués

Le fonctionnement des terminaux rencontrés peut être résumé par le pseudo-code suivant :

```
1  setInterrupt_InsertExtractCard();
2  setInterrupt_TimeOutCardKeyboard();
3
4  init();
5
6  while(1){
7      while( !isThereUSBCmd())
8      {
9          usbCmd = getUsbCmd();
10
11         if( isSecurePIN(usbCmd))
12         {
13             resp = doSecurePIN(usbCmd);
14         }
```

1. Les fabricants ont été informés des résultats obtenus sur leurs produits. Les failles découvertes ont été depuis corrigées.

```
15          else
16          {
17              apdu = extractAPDU();
18              resp = sendAPDU(apdu);
19          }
20              sendUsbResp(makeUsbResp(resp));
21      }
22  }
```

Listing 1. Pseudo-code du fonctionnement du terminal.

Il n'est donc pas nécessaire d'avoir un système temps réel complexe ou embarqué tel que Windows CE. C'est pourquoi ces terminaux reposent sur un simple micro-contrôleur de type Intel MCS-51 communément appelé 8051, très populaire à cause de son faible coût. Il pilote aussi un second micro-contrôleur qui permet de gérer l'affichage de l'écran LCD.

Les moyens d'exploiter un débordement de tampon vont être limités, puisque par exemple les notions de tas ou de thread n'existent pas. Le micro-contrôleur MSC-51 est constitué d'un coeur 8 bits. Il incorpore une architecture Havard (en opposition à l'architecture Von Neuman utilisé par exemple sur les processeurs de type x86). C'est à dire que le code et les données ne sont pas dans le même espace. Il contient généralement trois types de mémoire : le code, l'IRAM, et la XRAM. La taille physique réellement disponible pour chacune de ces mémoires est très réduite : moins de 100 Ko pour le code, moins de 2 Ko pour la XRAM et nécessairement 256 octets pour l'IRAM.

La pile est située dans l'IRAM, sa taille est donc inférieure à 256 octets. Il est rare d'avoir des tableaux locaux sur la pile, qui permettrait de l'écraser. Comme la taille des commandes d'une carte à puce est supérieure à 255 octets, le tableau les stockant est donc en XRAM. De plus, en raison de l'architecture de type Harvard, un tableau en XRAM, et par extension une réponse de la carte, ne pourra pas venir déborder sur la pile.

La possibilité d'écraser une adresse de retour est donc très rare et même si le cas se présente, elle serait difficile à exploiter. En effet, dans le cas d'une attaque en boîte noire, le code source n'est pas disponible. La seule façon de le retrouver est d'analyser physiquement la puce, ce qui est une opération longue et coûteuse. Cette caractéristique diminue fortement l'opportunité d'utiliser des attaques de type « Return-Oriented-Program » [14].

Nos chances d'exploitation d'un débordement vont reposer sur une autre particularité de cette architecture. Il s'agit de la possibilité d'indi-

quer lors de la compilation que les variables locales ne soient pas créées sur la pile, mais dans la XRAM, pour éviter de la saturer trop rapidement. Dans ce cas, un débordement de tampon pourrait éventuellement écraser en XRAM une variable globale ou bien une variable locale utilisée par une fonction appelante. Cela permet donc d'avoir un comportement incohérent dès que cette variable serait de nouveau utilisée. Cette configuration est très souvent utilisée. C'est le principal moyen d'exploitation que nous considérons comme valide.

6.2 Attaque par ATR

Dump d'une précédente transaction

Sur notre premier terminal, nous obtenons un comportement imprévu lorsque notre carte malveillante envoie un ATR codé avec un nombre d'octets historiques incohérent. Le terminal nous renvoie plus d'octets que prévus. Par exemple, avec l'ATR « 3B 03 », la carte n'émet que deux octets, alors que la structure indique que 5 octets doivent être normalement reçus (voir table 2). Néanmoins, le terminal ne vérifie pas la cohérence entre la structure et la taille lue ; il calcule que la réponse de la carte a une longueur de 5 octets et renvoie donc 3 octets de plus non-prévus.

En codant la taille maximale de champ, avec l'ATR « 3B 0F », le terminal peut renvoyer jusqu'à 15 octets non-prévus. D'autres ATR peuvent être construits de façon à lire plus de données, en utilisant les propriétés du quartet de poids fort des octets T0, TD1, TD2, et ainsi de suite. Au final, nous obtenons cinq ATR qui peuvent être exploités :

- 3B 0F,
- 3B FF 00 00 00 00,
- 3B FF 00 00 00 F0 00 00 00 00,
- 3B FF 00 00 00 F0 00 00 00 F0 00 00 00 00,
- 3B FF 00 00 00 F0 00 00 00 F0 00 00 00 F0 00 00 00 00.

Pour caractériser les données récupérées, avant d'effectuer notre attaque nous programmons notre émulateur pour qu'il renvoie une réponse correcte de 255 octets de 0x00 à 0xFE lors d'une commande particulière. Ensuite, nous déclenchons le renvoi d'un ATR mal structuré. Ces opérations sont renouvelées avec les différents ATR indiqués précédemment. Nous obtenons les réponses suivantes de la part du terminal :

- **3B 0F** *00 00 00 00 27 01 00 03* 00 01 02 03 04 05 06
- **3B FF 00 00 00 00** *0C 01 00 03* 00 01 02 03 04 05 06 07 08 09 0A

- **3B FF 00 00 00 F0 00 00 00 00** 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E
- **3B FF 00 00 00 F0 00 00 00 F0 00 00 00 00** 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12
- **3B FF 00 00 00 F0 00 00 00 F0 00 00 00 F0 00 00 00 00** 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16

Les caractères en gras correspondent à l'ATR envoyé par la carte. Les caractères en italique correspondent à l'entête d'une réponse CCID obtenue précédemment. Les caractères soulignés correspondent au champ de données de la réponse CCID. Entre le moment où le terminal renvoie la réponse de la carte et celui où il renvoie l'ATR, aucune commande CCID n'a été envoyée avec un champ de données. Cela veut dire que ces octets imprévus correspondent à une partie de la réponse de la carte utilisée précédemment. Cette hypothèse est confirmée par notre journal de tests.

Nous nous apercevons que :

- le tableau utilisé par le terminal pour récupérer l'ATR est le même que celui utilisé pour renvoyer une réponse CCID (entête et données).
- En refaisant le test en extrayant puis insérant la carte après avoir reçu la réponse correct, nous nous apercevons donc que ce tableau contenant les réponses CCID n'est pas nettoyé lorsqu'une nouvelle carte est utilisée. Il est possible de retrouver les 0x17 premiers octets de la dernière réponse de la précédente carte, contenus dans le champ de données de la réponse CCID

De l'importance d'effacer les données secrètes en mémoire volatile

Le résultat est légèrement différent quand le terminal attaqué vient d'être précédemment utilisé pour vérifier le code PIN de l'utilisateur via le « Secure Verify PIN » . En effet, l'utilisation de notre carte malveillante juste après cette opération avec l'ATR « 3B 0F » entraîne le renvoi des octets suivants par le terminal :

- **3B 0F** *00 00 00 00 D4 01 00 03* 00 00 00 00 00 00 00

Nous récupérons donc une suite d'octet à 0x00. Cet « écrasement » peut être le résultat d'une commande CCID envoyée ou reçue par le terminal entre la commande « Secure Verify PIN » et notre attaque. Cependant, notre journal de test infirme cette possibilité.

Une autre hypothèse peut être émise : il semble que le tableau de l'ATR soit le même que celui utilisé par le terminal pour construire la

commande « Verify PIN » envoyée à la carte. Néanmoins, comme ce tableau contient un bien critique, le code PIN, le terminal doit l'effacer. Une prise de contact avec le fabricant nous permettra de confirmer cette hypothèse.

Au final, cette faille est donc mineure puisqu'elle ne permet que de récupérer de courts fragments de la réponse d'une précédente transaction. De plus, le bien le plus sensible, le code PIN, est correctement effacé quand le terminal est utilisé en mode « Secure Verify PIN ». Cependant, une question demeure dans le cas où le tableau ne serait pas effacé, pourrions-nous réellement récupérer le code PIN ? Dans chaque réponse de notre l'ATR, les 10 premiers octets sont écrasés par un entête CCID. Nous simulons donc la construction de la structure de la commande « Verify PIN » et « Modify PIN ». Les octets écrasés par l'entête CCID sont en italique :

- *00 20 00 80 08 2C 12 34 56 78* **90 12 FF**. Il s'agit de la commande « Verify PIN », avec un PIN à 12 chiffres « 123456789012 », 0xFF est un octet de padding. Les quatre derniers chiffres pourraient être récupérés, mais un PIN à 12 chiffres n'existe pas à notre connaissance.
- *00 24 00 00 10 01 01 01 01 01* **01 01 01 02 02 02 02 02 02 02 02**. Il s'agit de la commande « Modify PIN ». L'ancien PIN est l'ensemble de 8 chiffres à « 1 », tandis que le nouveau PIN est l'ensemble de 8 chiffres à la valeur 2. Le nouveau PIN pourrait donc être récupéré.

Il serait donc possible de récupérer en partie le code PIN de l'utilisateur. Dans le cas d'un « Verify PIN », seuls les 4 derniers chiffres d'un PIN à seize chiffres peuvent être retrouvés. Il est très peu probable de voir un jour un code PIN utilisant une telle taille. Cependant, dans le cas du « Modify PIN », la totalité du nouveau code PIN pourrait être retrouvé. Ce mécanisme d'effacement est donc crucial pour la sécurité du produit.

6.3 Attaque par octets de procédure

Un dump de données via l'écran du terminal

Sur notre premier terminal, cette attaque s'effectue en mode T0 en essayant de perturber la machine état d'envoi de la réponse. Lors des tests, certaines commandes entraînent parfois un blocage du terminal, accompagné d'une corruption de l'écran. Le terminal doit être débranché pour fonctionner à nouveau normalement. Un débordement de tampon

est donc suspecté : le pointeur indiquant les données à afficher doit être écrasé.

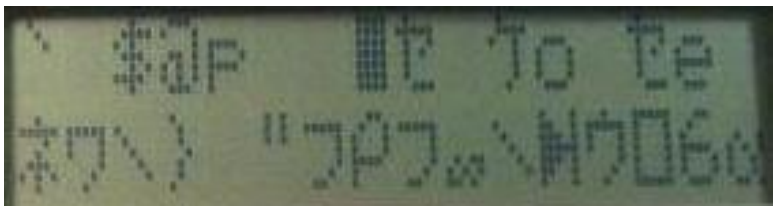

Figure 6. Exemple de corruption de l'écran.

Le code du terminal analysé est disponible. Néanmoins, l'envoi des commandes est géré nativement par le micro-contrôleur qui propose ce service via une bibliothèque. Son implémentation n'est pas disponible, seules les interfaces sont fournies. Le binaire du code du terminal ne contient pas le code de cette bibliothèque, juste l'appel à la fonction avec l'adresse adéquate. En effet la plupart de temps, les développeurs de micro-contrôleur ne fournissent que l'API de leur bibliothèque, ainsi que le point d'entrée des fonctions. Leur bibliothèque est codée en dur dans le micro-contrôleur, et donc inaccessible dans le but de protéger leur savoir-faire. L'analyse du code ne permet pas donc pas de caractériser le comportement.

Récupération du code applicatif

En exécutant des tests complémentaires, nous nous apercevons que les transactions effectuées avant l'attaque n'ont aucune influence sur la corruption de l'écran. De la même façon, l'utilisation du terminal en tant que simple lecteur de carte à puce ou dans un mode sécurisé n'entraîne pas de différences notables. L'affichage semble purement lié à la commande passée lors de l'attaque. Nous émettons l'hypothèse que la corruption de l'écran correspond à un dump de données statiques, par exemple, le code applicatif du terminal.

Les afficheurs possèdent une table de conversion valeur hexadécimale/caractères standards, qui est disponible sur Internet (voir la figure 7). Dans ce cas, il est donc possible de traduire l'affichage en caractères hexadécimaux. Presque tous les caractères peuvent être convertis, à part pour les éléments de 0x00 à 0x1F, ainsi que 0x80 à 0x9F. A partir du binaire du code, nous pouvons rechercher les octets convertis. La table 8 donne la

conversion de l'affichage du premier dump (les caractères intraduisibles sont notés XX).

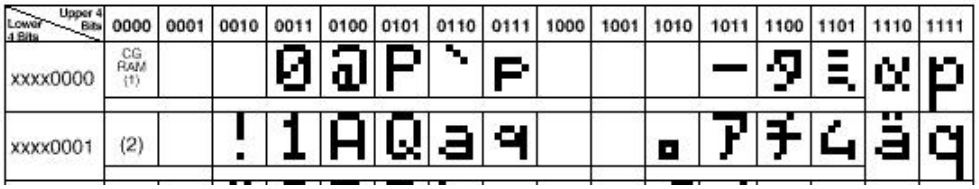

Figure 7. Table de conversion d'un caractère en hexadécimal.

Conversion
60 XX 24 40 70 XX XX 8E BE XX B9 6F XX BE 65 XX
C8 DC 12 7D XX 22 CC EF CC EC 12 1A B3 91 36 E0

Table 8. Conversion de l'affichage en caractères hexadécimaux.

Nous nous apercevons que chaque affichage converti peut être trouvé dans le fichier. Ce débordement de tampon permet donc de dumper le code de l'application via l'écran sous forme de caractères LCD. Le plus étonnant est que cette faille ne provient pas du code applicatif. Elle est issue de la bibliothèque de gestion de la communication de la carte à puce qui est fournie par le micro-contrôleur, mais dont le développeur n'a pas la maîtrise. Il n'a en effet à sa disposition que l'interface, et pas le code. Ce résultat démontre qu'en plus du code applicatif, il faut tester les bibliothèques du micro-contrôleur pour éviter de laisser une faille.

6.4 Attaque par APDU étendues en mode T1

Dump de la totalité de la RAM du composant

Sur notre deuxième terminal, l'envoi d'APDU étendues en mode T1 entraînent rapidement un comportement incohérent. La réponse n'est pas celle attendue, tant au niveau de la longueur que du contenu. Dans certains cas, le terminal ne fonctionne plus. Il fonctionne de nouveau normalement lorsqu'il est débranché puis remis sous tension. l'envoi d'une commande APDU non-étendue n'entraîne aucun problème. Une mauvaise gestion du mode étendu entraînant un débordement de tampon est donc suspecté. Les résultats qui vont être décrits ont été obtenus en boîte

noire, sans avoir le code à disposition. Les différentes interprétations ne sont donc que des hypothèses.

L'analyse des réponses récoltées permet de voir qu'elles contiennent des fragments de commandes ou de réponses de précédentes transactions. Des entêtes CCID sont aussi récupérés. Il semble que le débordement de tampon écrase des variables globales utilisées pour la gestion de la communication USB, plus précisément l'adresse, ainsi que la taille des données à renvoyer. Des tests complémentaires sont effectués. Les données renvoyées par notre charge sont modifiées, et nous essayons d'obtenir une corrélation avec les données renvoyées. Finalement, nous arrivons à forger une réponse dont deux octets permettent de contrôler l'adresse des données renvoyées. Chaque commande permet de récupérer 255 octets à chaque fois. Nous supposons que nous avons réussi à écraser la variable renseignant l'adresse ainsi que la longueur des données à renvoyer. Finalement, nous arrivons à récupérer 2047 (0x7FF) octets. Au-delà de cette adresse, nous ne récupérons que des octets à 0x00, sans avoir néanmoins un blocage du terminal.

Récupération du code PIN d'une précédente transaction

Dans le but de caractériser ces données, nous recommençons notre attaque en effectuant auparavant une transaction correcte :

- Le terminal envoie une commande de 261 octets à la carte. Elle est correctement acceptée par le terminal puisque ce n'est pas une APDU en mode étendue. Elle est notée CTPDU, et vaut : « 80 AF FF FF FF 00 BA 01 BA 02 BA 03 BA 04 BA 05 [. . .] BA 7E BA 7F 00 ».
- La carte renvoie ensuite une réponse valide de 257 octets. Elle est notée RTPDU, et vaut : « 00 44 55 4D 50 54 45 53 54 01 44 55 4D 50 54 45 53 54 ... 90 00 ».
- Nous effectuons ensuite l'attaque en envoyant la commande « 80 A8 00 00 ».
- Celle-ci va déclencher l'envoi par la carte de notre charge. La réponse a été configurée pour déclencher le dump des 255 premiers octets. Nous le récupérons ensuite.

L'ensemble de cette manipulation est refaite jusqu'à dumper les 0x7FF octets (à chaque fois, nous modifions la valeur de notre charge pour décaler l'adresse du dump). Au final, trois éléments principaux sont identifiés dans les données récupérées (voir la table 9) :

- Les 261 premiers octets correspondent au tableau de réception de la commande CCID. Il y a la commande CTPDU envoyée avant notre

attaque. Les quatre premiers octets sont écrasés par la commande permettant de déclencher notre charge.
– Les 271 octets suivants semblent correspondre à la réponse CCID. En effet, nous trouvons un entête caractéristique, ainsi que la réponse précédente de la carte.
– Nous voyons plus loin une partie de la commande CAPDU et de la réponse RAPDU précédente. Nous supposons que l'applicatif ou le micro-contrôleur effectue des copies avant l'envoi, puis après la réception de la commande de la carte.

Les mêmes observations s'appliquent lorsque nous modifions les valeur de CTPDU, RTPDU ou de la commande déclenchant la charge, ce qui valide notre interprétation.

Après nous être servi du terminal comme simple lecteur de carte à puce, nous effectuons notre attaque lorsque celui a été précédemment utilisé en mode « Secure Verify PIN » ou « Secure Modify PIN ». En effet, le PIN est saisi sur le clavier du terminal, puis ce dernier construit une commande APDU le contenant. Ces éléments doivent être obligatoirement écrits en mémoire à un instant donné. Néanmoins nos tests montrent que le code PIN n'est pas présent dans le dump. Nous supposons qu'il est soit effacé par le terminal, ou soit écrasé par la réponse de notre carte malveillante.

Un point important demeure, nous voyons que le tableau de réception et d'émission de la commande CCID n'est pas nettoyé entre chaque commande ou lors de l'extraction ou de l'insertion d'une carte. Avec une carte malveillante, il est donc possible de récupérer la commande ainsi que la réponse de la précédente transaction et donc les éventuelles données sensibles qu'elles contiennent. Néanmoins, les quatre premiers octets de la commande sont écrasés par notre charge, c'est la taille minimale qui peut être utilisée. Cependant, si une commande « Verify PIN » ou « Modify PIN » est envoyée en clair au terminal, nous pouvons quand même récupérer le code PIN (voir la table 9). Bien sûr, nous ne mettons pas en défaut le mode sécurisé. Cependant, si des données secrètes sont échangées en clair, une grande partie d'entre elles peut être récupérée. Il est donc déconseillé de manipuler des données sensibles en clair dans les commandes APDU.

La mémoire XRAM pourrait être éventuellement modifiée d'une autre façon pour altérer le fonctionnement du produit. Nous n'avons pas pu la mettre en oeuvre dans le temps imparti. Après une prise de contact avec les développeurs, plus tard que les communications effectuées par le terminal s'appuyait sur une bibliothèque fournie par le micro-contrôleur.

Adresse	Données
0x0000	80 A8 00 00 FF 00 BA 01 BA 02 BA 03 BA 04 BA 05 BA 06 BA 07 BA 08 BA 09 BA 0A BA 0B BA 0C BA 0D [...] BA 76 BA 77 BA 78 BA 79 BA 7A BA 7B BA 7C BA 7D BA 7E BA 7F 00
0x0105	80 02 00 00 00 00 11 00 00 00 55 4D 50 54 45 53 54 01 44 55 4D 50 54 45 53 54 02 44 55 4D 50 54 45 53 54 03 44 55 4D 50 54 45 53 [...] 25 44 55 4d 50 54 45 53 54 26 44 55 4d 50 54 45 00 00 00 00 01 7e 02 00 00 00 00 00 00 00 00 00
0x0350	[...] 00 00 00 00 00 00 00 00 00 00 00 00 90 00 9F 53 54 24 44 55 4D 50 54 45 53 54 25 44 55 4D 50 54 45 53 54 26 44 55 4D 50 54 45 53 54 27 44 55 4D 03 00 17 BA 1C BA 1D BA 1E BA 1F BA 20 BA 21 BA 22 BA 23 BA 24 BA 25 BA 26 BA 27 BA 28 BA 29 BA 2A BA 2B BA 2C BA 2D BA 2E BA 2F BA 30 BA 31 BA

TABLE 9. Interprétation des données dumpées.

Adresse	Données
0x0000	80 A8 00 00 08 24 12 34 FF FF FF FF FF 04 BA 05 BA 06 BA 07 BA 08 BA 09 BA 0A BA 0B BA 0C BA 0D

TABLE 10. Récupération du code PIN.

Pourtant, cette bibliothèque spécifiait clairement qu'elle se protégeait elle-même des débordements de tampon. En effet, en cas de détection d'un débordement de tampon, elle devait renvoyer un statut d'erreur. C'est pourquoi le développeur n'avait pas implémenté de contre-mesures spécifiques. Encore une fois, la faille provient du micro-contrôleur.

7 Conclusion

De simples lecteurs de carte à puce se branchant en USB et dotés d'un écran et d'un clavier permettent une saisie sécurisée du code PIN d'une carte à puce dans le but d'utiliser ses services via une application installée sur un PC. Cet article s'est intéressé à la sécurité de ces terminaux du point de vue des attaques logicielles via l'interface de la carte à puce.

La mise en place de ces attaques a nécessité la création de notre propre émulateur de carte à puce à base d'Arduino puisque les solutions disponibles sur le marché n'étaient pas totalement satisfaisante. Son pilotage via un serveur python sur notre PC nous a permis de l'intégrer facilement dans notre solution logicielle pour nous permettre de faire des tests de fuzzing.

Cet article a pu démontrer que malgré son périmètre restreint, la couche protocolaire ISO/IEC 7816-3 est un vecteur d'attaque à prendre en compte. En effet, plusieurs failles basées sur notre analyse de vulnérabilité ont été découvertes : l'ATR, les octets de procédures en mode T0, ainsi que les APDUs étendues en mode T1. De plus, chaque faille a pu être exploitée. Les faits les plus probants sont liés à l'exploitation de débordements de tampon entraînant un effet inattendu : le dump du code applicatif du terminal sur son propre écran. Un autre effet a été le dump de la la RAM, permettant de lire la totalité d'une précédente transaction, dont le code PIN dans certain cas (même si cela n'était pas en mode sécurisé). Ces deux derniers résultats sont très intéressants, car ils ne reposent pas sur une erreur de programmation des développeurs du terminal, mais sur une faille des bibliothèques applicatives du micro-contrôleur sous-jacent utilisé.

Ces résultats démontrent encore une fois l'importance de principes de base comme l'effacement des données ou des secrets entre chaque transaction. Un autre point est de ne pas faire confiance au matériel sous-jacent et de le tester. En effet, si la compréhension et l'exploitation des failles ont été complexes, de simples tests auraient pu rapidement mettre en évidence un comportement incohérent.

Notre approche est applicable à tout terminal communicant avec une carte à puce en contact (en mode T0 ou T1). Nous pouvons citer les terminaux bancaire ou à génération de mot de passe unique, et même les téléphones avec leur carte SIM (il suffit juste de changer la raquette de branchement). Leur architecture étant souvent plus complexe (Windows CE, Linux), les possibilités de découvrir et d'exploiter des failles ne sont que plus importantes.

Références

1. Deja vu security, peach fuzzing platform. `http://www.peachfuzzer.com`.
2. PINGUINO, open source hardware electronics prototyping platform based on 8- and 32-bit pic from microchip. `http://www.pinguino.cc`.
3. RASPBERRY PI. `http://www.raspberrypi.org`.
4. Consortium. *Universal Serial Bus, Device Class : Smart Card CCID, Specification for Integrated Circuit(s) Card Interface Devices*, Revision 1.1, April 22rd, 2005.
5. Aitel D. The advantages of block-based protocol analysis for security testing. *Immunity Inc.*, Février 2012.
6. GIXEL. Plateforme commune pour l' eadministration — spécification technique. *Révision : 1.01*, Révision : 1.01, Novembre 2006.
7. INTERNATIONAL STANDARD ORGANIZATION FOR STANDARDIZATION (ISO). *Identification cards — Integrated circuit(s) cards with contacts — Part 3 : Electronic signals and transmission protocols*, 2006-11-01.
8. INTERNATIONAL STANDARD ORGANIZATION FOR STANDARDIZATION (ISO). *Identification cards — Integrated circuit(s) cards with contacts — Part 4 : Organization, security and commands for interchange*, 2006-11-01.
9. MasterCard. M/chip 4 card application specifications for debit and credit - part a. *Version 1.1*, Septembre 2006.
10. Tilo Müller, Tobias Latzo, and Felix C. Freiling. Hardware-based full disk encryption (in)security. *Chaos Computer Club's 29th hacker conference*, 2012.
11. Nils and Rafael Dominguez Vega. PINPADPWN. *BlackHat*, 2012.
12. Amini P. and Protnoy A. Sulley fuzzing framework. `http://code.google.com/p/sulley`.
13. Godefroid P., Levin M.Y., and Molnar D. Automated whitebox fuzz testing. *NDSS*, 2008.
14. Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming : Systems, languages, and applications.
15. Ludovic Rousseau. CCID free software drive. `pcsclite.alioth.debian.org/ccid.html`.
16. The H Security. CCC reveals security problems with German electronic IDs. `http://h-online.com/-1094577`, 22 septembre 2010.
17. Ganesh V., Leek T., and Rinard. M. Taint-based directed whitebox fuzzing. *IEEE 31st International Conference on Software Engineering*, 2009.

18. VISA. Visa integrated circuit card specification (vis). *Version 1.5*, May 2009.
19. PC/SC Workgroup. *Interoperability Specification for ICCs and Personal Computer Systems. Part 1. Introduction and Architecture Overview*, Revision 2.01.01, September 2005.
20. PC/SC Workgroup. *Interoperability Specification for ICCs and Personal Computer Systems, Part 10 IFDs with Secure PIN Entry Capabilities*, Revision 2.02.08, April 2010.