Un framework de fuzzing pour cartes à puce : application aux protocoles EMV

Julien Lancia

SERMA Technologies, CESTI, 30, avenue Gustave Eiffel 33600 Pessac, France
j.lancia(@)serma.com

Résumé Les cartes à puce constituent des plateformes pour lesquels la sécurité est un enjeu majeur, c'est pourquoi elles offrent des mécanismes de protection avancés au niveau matériel. Cependant, la sécurité globale de ces plateformes peut être mise en défaut par une vulnérabilité au niveau applicatif. Ce papier présente notre framework de fuzzing pour cartes à puce qui permet de tester en profondeur les implémentations de protocoles dédiés aux cartes à puce et d'identifier les vulnérabilités présentes dans ces implémentations. A l'aide de notre framework, nous avons implémenté deux fuzzers pour les protocoles de paiement sécurisé par cartes à puce les plus répandus dans le monde. Les résultats obtenus par fuzzing sur des cartes à puce du marché démontrent l'adéquation de l'approche de test par fuzzing au domaine des cartes à puce.

1 Introduction

Les cartes à puce remplacent progressivement les cartes magnétiques pour les applications de paiement et de retrait dans de nombreux pays. Avec l'émergence des applications d'e-administration, la croissance continue du marché de la téléphonie mobile nécessitant des cartes SIM, et la multitude d'applications allant du micro-paiement à la santé, le nombre de cartes à puce déployées à la fin de l'année 2009 est estimé à 8 milliards [1], soit plus que le nombre d'habitants sur terre. La sécurité offerte par ces plateformes, ainsi que leur simplicité d'utilisation, nous ont conduit à confier progressivement une part importante de nos données personnelles et confidentielles à ces plateformes matérielles. Il est donc crucial de s'assurer du bon fonctionnement tant sécuritaire que fonctionnel des applications embarquées sur ces plateformes.

Afin de répondre à cette problématique, nous proposons un framework de fuzzing [1] permettant de tester et rechercher les failles dans les protocoles dédiés aux cartes à puce. Nous nous intéresserons en particulier au protocole EMV [3,4,5,6], qui est actuellement le système de paiement par carte à puce le plus répandu dans le monde.

La suite de cet article est construite de la manière suivante. Après une présentation du protocole EMV et des défis posés par son implémentation, nous introduirons les

1. Les termes fuzzing, fuzzer et framework n'ont actuellement pas de traduction française satisfaisante. Nous utilisons donc les termes anglais dans cet article.

techniques de test par fuzzing et leur adéquation à la recherche de failles dans les protocoles dédiés aux cartes à puce.

Nous présenterons ensuite les efforts mis en œuvre afin d'évaluer la robustesse des implémentations de ces protocoles, et le framework de fuzzing pour cartes à puce résultant.

Nous détaillerons l'instanciation de notre framework pour tester les protocoles M/Chip et VIS (protocoles propriétaires dérivés de EMV).

Finalement, nous exposerons les résultats obtenus par l'application de nos fuzzers sur des produits du marché.

2 Adéquation des techniques de fuzzing aux protocoles pour cartes à puce

Les spécifications EMV (nom tiré de ses membres historiques Europay, MasterCard et Visa) de l'organisation EMVCo constituent le protocole de paiement le plus déployé dans le monde. Au début de l'année 2008, on comptait 730 millions [2] de cartes de paiement compatibles EMV dans le monde. De par sa conformité aux normes ISO/IEC 7816 encadrant les communications des systèmes à cartes à puce [7,8], et son déploiement mondial, ce protocole est représentatif des protocoles pour cartes à puce et est donc utilisé comme cas d'étude tout au long de cet article.

2.1 Les Spécifications EMV

Présentation EMV constitue un cadre général pour la définition de transactions de paiement sécurisé. Quatre documents principaux constituent les spécifications :

- Livre 1 : Interface entre la carte et le terminal, indépendante de l'application,
- Livre 2 : Sécurité et gestion des clés,
- Livre 3 : Spécification de l'application,
- Livre 4 : Interface du porteur de carte, du commerçant et de l'acheteur.

Nous nous intéresserons plus particulièrement ici aux spécifications de l'application, détaillées dans le « Livre 3 ».

EMV constitue à la fois un ensemble de protocoles et un cadre à partir duquel des protocoles propriétaires peuvent être développés. Les règles énoncées dans ces spécifications imposent des fonctionnalités minimales à mettre en œuvre ainsi que la façon de les implémenter pour évoluer dans un cadre EMV. Les éléments non imposés par la spécification peuvent être définis librement dans le cadre d'un protocole propriétaire (par les systèmes de paiement comme VIS [9], propriétaire VISA, ou M/Chip [21], propriétaire MasterCard).

Le déroulement d'une transaction EMV est composé de 8 étapes que nous détaillons ici. La figure 1 représente une vue haut niveau d'une transaction EMV complète, ainsi que les commandes APDU (Application Protocol Data Unit) échangées entre le terminal et la carte.

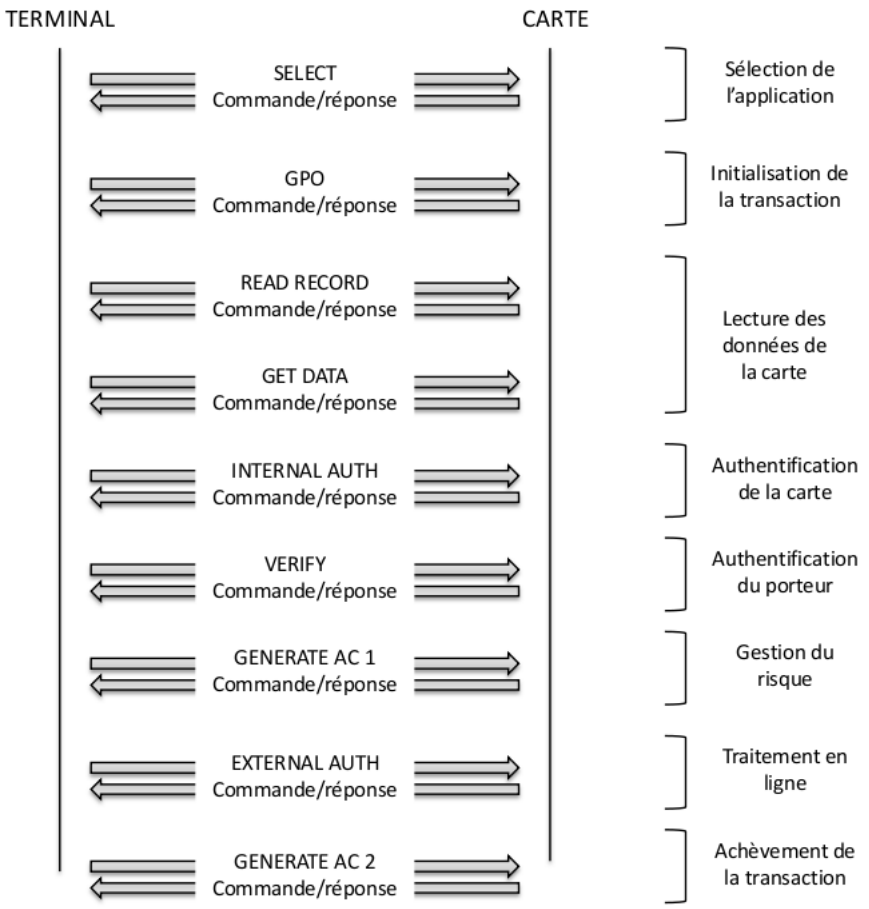

Figure 1. Transaction EMV type.

Sélection de l'application. La commande *SELECT* permet au terminal de sélectionner une application présente sur la carte par son identifiant unique, l'AID (Application Identifier).

Initialisation de la transaction. La commande *GET PROCESSING OPTIONS* (GPO) permet d'informer la carte du commencement d'une transaction, et d'échanger les informations nécessaires pour débuter la transaction.

Lecture des données de la carte. Cette étape permet au terminal de récupérer les informations de la carte. Le terminal envoie une succession de commandes *READ RECORD* ou *GET DATA* et stocke les données ainsi obtenues pour une utilisation ultérieure.

Authentification de la carte. L'authentification de la carte peut se faire de manière statique ou, pour assurer un niveau de sécurité supérieur, de manière dynamique.

Dans le cas de l'authentification statique, le terminal réalise un processus de signature cryptographique à clé publique de données statiques lues sur la carte afin de valider l'authenticité de cette dernière.

L'authentification dynamique est réalisée en utilisant la commande *INTERNAL AUTHENTICATE* et permet au terminal d'authentifier la carte dynamiquement : celle-ci doit cette fois recalculer une nouvelle signature à chaque transaction.

Authentification du porteur de carte. Le porteur indique son code PIN (Personal Identification Number) au terminal qui l'envoie à la carte via la commande *VERIFY*.

Gestion du risque par la carte et le terminal. A ce stade, le terminal doit prendre une première décision pour l'acceptation de la transaction (avec ou sans connexion vers la banque émettrice), ou le rejet. Ceci évite de prolonger une transaction dont l'issue serait négative dans tous les cas.

Une fois ces paramètres examinés, le terminal envoie à la carte une première commande *GENERATE APPLICATION CRYPTOGRAM*, qui représente le cœur de la transaction proprement dite et reflète la décision prise par le terminal.

Le champ de données envoyé par cette commande contient entre autres le montant de la transaction, la monnaie utilisée et les capacités du terminal. La réponse de la carte indique au terminal la décision pour l'acceptation de la transaction (« offline »), la demande de connexion (« online »), ou le rejet.

Traitement en ligne. S'il y a lieu (la carte a détecté que la transaction était hors de certaines limites fixées par l'émetteur, et s'en remet à lui pour délivrer une décision définitive), la transaction est envoyée en ligne à l'émetteur. La carte doit authentifier l'émetteur pour éviter toute fraude, ce qui peut être réalisé lors d'une seconde commande *GENERATE APPLICATION CRYPTOGRAM* ou traité par le biais de la commande *EXTERNAL AUTHENTICATE*.

Achèvement de la transaction. C'est la phase de clôture de la transaction au cours de laquelle la carte renvoie une décision définitive (acceptation ou rejet de la transaction) via une seconde commande *GENERATE APPLICATION CRYPTOGRAM*.

Limitations Les spécifications EMV sont issues de la coopération entre trois organismes de paiement internationaux. Ces spécifications ont été conçues comme un cadre d'interopérabilité censé permettre le développement de protocoles propriétaires, aussi EMV offre une grande souplesse dans le choix des options d'implémentation tant fonctionnelles que sécuritaires. Les protocoles propriétaires développés dans le cadre EMV offrent un raffinement de cette multitude d'options, mais certains [9] conservent de nombreuses fonctionnalités optionnelles qui rendent le développement d'applications sujet aux erreurs.

Par ailleurs, ces spécifications sont extrêmement volumineuses. L'ensemble des spécifications EMV représente un volume de 700 pages, et les détails d'implémentation pour une commande donnée sont répartis sur 2 à 3 livres distincts. Par exemple, l'implémentation de la commande VERIFY permettant d'authentifier le porteur de la carte nécessite les informations suivantes :

- Format de la commande, EMV Livre 3 chapitre 6,
- Condition et séquence d'exécution, EMV Livre 3 chapitre 10,
- Actions du terminal, EMV Livre 4 chapitre 6,
- Chiffrement du PIN, EMV Livre 2 chapitre 7,
- Code des algorithmes autorisés, EMV Livre 2 Annexe B,
- Protocoles de transmission, EMV Livre 1.

En conséquence, la nature même de ces spécifications induit une probabilité élevée d'introduire des failles sécuritaires ou fonctionnelles dans les implémentations. Il est donc nécessaire de mener des campagnes de tests aussi complètes et exhaustives que possibles sur ces produits afin d'identifier le plus tôt possible les failles existantes. Comme présenté par la suite, les techniques de test par fuzzing s'avèrent particulièrement adaptées pour répondre à ce besoin.

2.2 Recherche de vulnérabilités par fuzzing

Technique de test par fuzzing Le fuzzing est une des techniques les plus simples et efficaces pour détecter des vulnérabilités dans les implémentations logicielles. Le fuzzing est généralement décrit comme une technique de test en boîte noire (bien que des travaux de fuzzer en boîte blanche existent [10,11]) dans laquelle des données sont générées et soumises au système testé afin d'éprouver sa robustesse. Cette technique permet d'explorer un grand nombre de cas de test de manière automatique, ce qui serait fastidieux à réaliser par des tests manuels.

Les outils permettant de réaliser des tests par fuzzing sont de 2 types. Les fuzzers spécifiques, qui permettent de tester un unique programme ou protocole, et les frameworks de fuzzing. Ces derniers proposent une interface de programmation permettant d'implémenter des fuzzers pour une famille d'applications ou de protocoles.

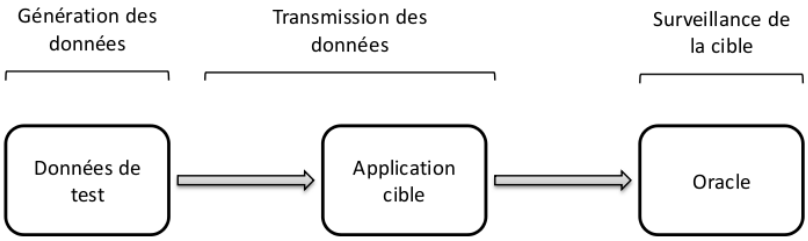

Figure 2. Modèle de fuzzer.

D'une manière générale, la structure d'un outil de fuzzing est toujours identique [12]. La figure 2 présente un modèle générique de fuzzer. Il est composé de 3 grandes fonctions que sont :

- la génération de données (créer les données qui seront envoyées à la cible)
- la transmission des données (acheminer les données générées à la cible)
- la surveillance de la cible et l'audit (observer et enregistrer les réactions de la cible)

Bien que la majorité des fuzzers soit dédiée à la recherche de vulnérabilités dans les implémentations de protocoles réseaux [13,14,15], on trouve de nombreuses autres applications de la technique de fuzzing. Ces applications couvrent entre autres les applications en ligne de commande [22], les formats de fichiers [23] ou les communications interprocessus [24]. A notre connaissance, il n'existe pas d'outil de fuzzing permettant de tester les protocoles de cartes à puce.

Application aux cartes à puce Les techniques de test par fuzzing sont pourtant bien adaptées au domaine des cartes à puce, et en particulier aux protocoles EMV. En effet, les APDU sont composées de champs bien déterminés pouvant prendre une large gamme de valeurs. L'exploration manuelle de ces différentes valeurs s'avère fastidieuse, et le test exhaustif de toutes les combinaisons de valeurs mène à une explosion combinatoire. La fonction de génération automatique de données de test proposée par la technique de fuzzing est donc pertinente.

D'autre part, comme présenté dans la section 0, les spécifications des protocoles pour cartes à puce sont souvent volumineuses et complexes. Il est donc difficile de détecter des failles par une analyse du code source, et plus particulièrement pour des failles subtiles résultant de choix d'options d'implémentation. Le fuzzing, qui permet

d'explorer un protocole en profondeur tant pour des données valides, qu'invalides, permet de mettre en lumière ces failles.

Enfin, le code source des applications EMV est considéré comme un bien à protéger et est donc rarement disponible lors des tests. Une technique de test en boîte noire telle que le fuzzing est donc bien adaptée.

Nous orientons notre choix technique vers un framework de fuzzing plutôt que vers un fuzzer spécifique, ce qui nous permet d'exploiter notre fuzzer sur l'ensemble des protocoles dédiés aux cartes à puces, qui possèdent de grandes similitudes. En particulier, nous nous intéressons aux techniques de fuzzing par bloc que nous détaillons par la suite.

3 Le fuzzing par bloc appliqué aux cartes à puce

Historiquement, le test d'implémentations de protocoles consistait à développer un client pour un protocole, et à appliquer des modifications que le testeur pensait de nature à provoquer des erreurs dans le système cible. Cette technique de test possède les limitations suivantes :

- Le testeur doit émettre des hypothèses sur les modifications susceptibles de provoquer des erreurs,
- Le développement d'un tel client est souvent complexe, et l'outil ainsi obtenu n'est pas portable pour d'autres protocoles, fussent-ils de la même famille.
- L'architecture de fuzzer par bloc apporte une solution à cette problématique.

3.1 Le fuzzing par bloc

Présentation Lorsque le testeur dispose de la spécification, le fuzzing par bloc est une technique de test particulièrement efficace. Cette architecture est basée sur le constat que la plupart des protocoles sont construits sur le même modèle : les trames ou commandes du protocole sont formées de plusieurs champs, dont certains sont récurrents, tels que la longueur de trame, la longueur d'un champ de donnée ou les éléments de padding. Généralement, l'enchaînement valide des différentes trames est également normalisé par le protocole.

L'architecture de fuzzer par bloc consiste à modéliser la structure des données échangées dans le cadre du protocole. Dans un premier temps, des éléments de données sont assemblés en séquence pour former des blocs qui modélisent les trames du protocole. Ces blocs sont ensuite liés entre eux pour former un graphe, modèle complet du protocole.

A partir de ce modèle, le fuzzer génère automatiquement les données de test conformément à la structure décrite.

La puissance de cette architecture réside dans le fait que les tests portent non seulement sur les données de chaque trame, mais aussi sur l'enchaînement des trames. Le protocole peut ainsi être testé en profondeur, c'est à dire sur de larges gammes de valeurs pour différents enchaînements de trames.

Le fuzzer SPIKE [13], considéré comme le pionnier de ce type d'architecture, a inspiré les auteurs de travaux plus récents tels que Sulley [15] et Peach [14].

Ces outils sont dédiés au test de protocoles réseaux, mais peuvent servir de base pour les protocoles de cartes à puce qui comportent de nombreuses similitudes avec les protocoles réseaux :

- Les APDU, comme les trames des protocoles réseaux, sont découpés en champs bien déterminés,
- L'enchaînement valide des commandes est normalisé par la spécification,
- Les champs récurrents que sont la longueur d'APDU, la longueur des champs et les éléments de padding sont présents dans les deux cas.

Nos travaux se basent donc sur des outils existants, que nous adaptons pour les étendre au domaine de la carte à puce.

Choix techniques Notre choix pour le développement d'un framework de fuzzing pour carte à puce s'est porté sur le framework Sulley. Ce choix technique est motivé par plusieurs facteurs :

- Cet outil est déjà éprouvé dans le milieu industriel pour le test de protocoles réseaux [16],
- Il est implémenté en Python [17], qui est le langage dans lequel nos outils métiers sont implémentés, ce qui facilite la phase d'intégration,
- Il est reconnu comme l'un des frameworks de fuzzing les plus complets et efficaces [18,19,20].

En termes de génération de données, le framework Sulley offre plusieurs types élémentaires permettant de construire des trames (le lecteur est invité à se référer à la documentation de Sulley pour une liste exhaustive des types de données) :

- *static* – Chaîne de caractère invariable durant le test,
- *bit_field* – Représentation binaire d'une valeur entière. Afin d'éviter une explosion combinatoire lors des tests, toutes les valeurs possibles pour le champ ne sont pas utilisées pour générer les données de test. Les valeurs de test sont construites sur un intervalle $[+10, -10]$ autour de valeurs représentatives (valeur maximale divisée par 2, 4, 8,...) pour former jusqu'à 141 cas de test par champ,
- *size* – Taille d'un champ donné.

Des fonctionnalités additionnelles liées à la génération de données sont également offertes par le framework, telles que les encodeurs qui permettent d'associer un

traitement spécifique à certains champs (par exemple un chiffrement cryptographique) ou les callbacks qui permettent de modifier dynamiquement les données à envoyer en fonction de la valeur des réponses reçues précédemment.

En ce qui concerne la transmission de données, Sulley étant un framework de fuzzing orienté protocole réseau, la transmission est implémentée par un mécanisme de sockets.

Afin d'assurer la surveillance de la cible, l'application cible (le serveur) est exécutée dans un débogueur. Ainsi, si les données générées par le fuzzer provoquent un arrêt inattendu de l'application, le débogueur enregistre les circonstances de l'arrêt et redémarre l'application pour continuer le test. Par ailleurs un observateur réseau enregistre les trames circulant sur le réseau afin de compléter les données d'audit.

Notre approche consiste à adapter le framework Sulley au domaine des cartes à puce, principalement au niveau des fonctionnalités de transmission de données et de surveillance de cible.

3.2 Un framework de fuzzing pour cartes à puce

Au niveau applicatif, les données échangées entre les cartes à puce et les terminaux (les APDU) sont une suite d'octets au format hexadécimal. Le type élémentaire *bit_field* offert par Sulley permet donc, moyennant une mise en forme avant l'émission, de représenter la plupart des champs des APDU. Certains champs des APDU sont au format BCD (Binary Coded Decimal), qui est une représentation hexadécimale d'un chiffre décimal telle que les deux apparaissent visuellement identiques. Encore une fois, ces champs peuvent être obtenus par une mise en forme du type élémentaire *bit_field* offert par Sulley. Enfin, les longueurs de champs au format hexadécimal qui sont récurrents dans les APDU sont fournies par le type *size* du framework Sulley.

La génération de données étant prise en charge par le framework Sulley, le principal effort d'adaptation de ce framework au domaine de la carte à puce porte sur la transmission de données et la surveillance de la cible. La figure 3 présente l'architecture de notre Fuzzer pour cartes à puce.

Transmission de données Afin de permettre l'envoi des données de test générées par le fuzzer aux cartes à puce, nous avons intégré un outil métier, Triton, au framework de fuzzing. Cette intégration est simplifiée par la structure modulaire du framework Sulley, et par le fait que les deux outils sont développés dans le langage Python.

L'interface standard pour communiquer avec les cartes à puce est définie par les normes ISO 7816. Au niveau applicatif, cette communication s'appuie sur les commandes APDU. Pour pouvoir envoyer les commandes APDU à la carte, via un lecteur, l'interface standard PCSC [29] est utilisée. La technologie PCSC présente

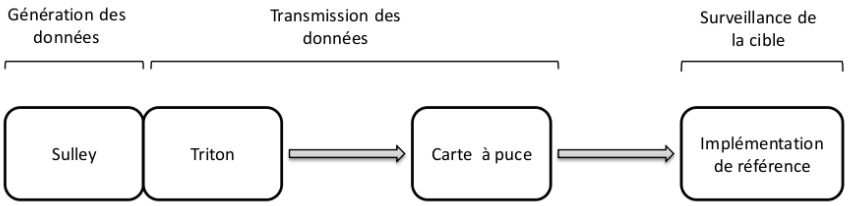

Figure 3. Architecture du fuzzer pour cartes à puce.

deux avantages : elle est portable sur plusieurs systèmes d'exploitation, et permet aux applications de faire abstraction des protocoles de communication propres à chaque lecteur.

Triton est une bibliothèque Python de communication avec les cartes à puce via l'interface standard PCSC. Triton est donc un ensemble de classes, de modules et de bibliothèques qui étendent le langage Python par des commandes de communication avec la carte, de gestion des APDU, de fichiers d'audit, de fonctions cryptographiques et de manipulation de données.

Dans le module de transmission de données de Sulley, nous avons remplacé le système de communication réseau basé sur les sockets par un système de communication PCSC basé sur la bibliothèque Triton. Ce nouveau système de transmission de données nous permet d'envoyer les données générées par le fuzzer à un terminal qui se charge de les envoyer à la carte, et de recevoir les réponses de la carte afin d'assurer la surveillance.

Surveillance de la cible Contrairement aux applications cibles du framework Sulley (des serveurs implémentant des protocoles réseaux), les applications s'exécutant sur cartes à puce ne peuvent être facilement exécutées en mode débogage. En effet, les applications sont fortement dépendantes des plateformes matérielles sur lesquelles elles s'exécutent, aussi les débogueurs sont généralement des outils propriétaires qui sont difficiles, voire impossible à intégrer dans des outils tiers.

Notre approche pour détecter les anomalies provoquées par l'envoi de données de test à la cible est basée sur une implémentation de référence du protocole testé jouant le rôle d'oracle. La figure 4 présente l'approche que nous avons mise en œuvre pour la surveillance de la cible.

Les données envoyées à la carte sont simultanément envoyées à une implémentation de référence du protocole. Une anomalie est détectée lorsque le résultat renvoyé par la carte diffère de celui renvoyé par l'implémentation de référence. Lors de la détection

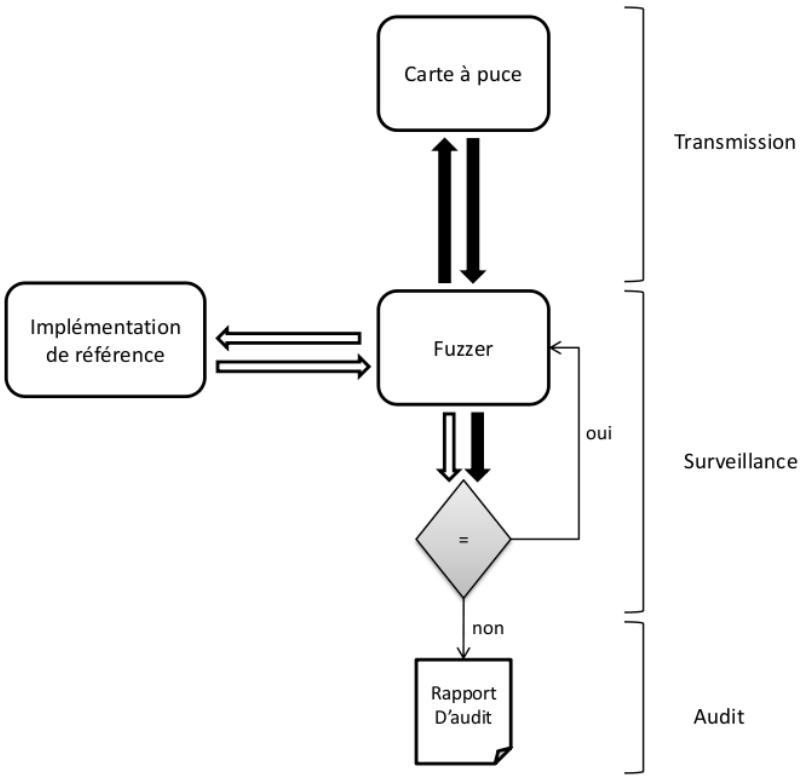

Figure 4. Surveillance de la cible.

d'une anomalie, le fuzzer enregistre les données ayant provoqué l'anomalie afin de les inclure dans le rapport d'audit.

Pour que cette approche soit réalisable, il est nécessaire que l'application testée et l'implémentation de référence possèdent la même configuration interne. Cette configuration est déterminée automatiquement par l'implémentation de référence avant de débuter les tests grâce aux informations lues sur la carte, et aux informations renvoyées lors de la transaction. Afin d'exécuter un fuzzing complet du protocole, il est toutefois nécessaire de connaître un certain nombre d'informations secrètes de la carte telles que le code PIN du porteur ou la clé privée permettant la génération de cryptogrammes.

Lorsque ces données ne sont disponibles, ou pour tester certaines commandes dont le contenu n'est pas déterminant, nous proposons une autre approche de surveillance.

Notre framework permet d'associer à chaque bloc du fuzzer une fonction (appelée Verifier) qui a pour rôle de valider les données renvoyées par la carte. Ces fonctions sont définies par le développeur lors de l'implémentation d'un fuzzer pour un protocole particulier. Lorsqu'une fonction Verifier détecte un retour anormal, elle renvoie un message d'erreur permettant d'identifier la nature et la cause de l'erreur. Ce message est enregistré par le fuzzer pour la génération du rapport d'audit. A titre d'exemple, ce type d'approche est exploité dans le cadre du fuzzer EMV pour la commande GET DATA. Cette commande permet de récupérer des données de la carte en spécifiant un identifiant unique et normalisé par la spécification (« tag »). Le test dans ce cas consiste à explorer l'ensemble des valeurs d'identifiants possibles afin de détecter des retours de données pour des identifiants non normalisés. Il serait laborieux d'initialiser l'implémentation de référence avec l'ensemble de ces valeurs, d'autant plus que la valeur elle-même a peu d'importance dans ce cas. Cette commande (ou bloc dans le cadre du fuzzer) est associée à une fonction Verifier qui assure que la commande ne renvoie des données que pour les tags normalisés.

Notre framework de fuzzing est générique et permet de tester les protocoles respectant les normes ISO encadrant les communications des systèmes à cartes à puce. Afin de valider notre approche, nous avons réalisé deux instances de fuzzer pour des protocoles EMV propriétaires.

4 Le fuzzing des protocoles EMV

Nous avons exploité notre framework de fuzzing pour réaliser deux fuzzers pour les protocoles propriétaires M/Chip et VIS. Les détails sur ces fuzzers ainsi que les résultats obtenus sont présentés ci-dessous.

4.1 Les fuzzers EMV

Les protocoles EMV propriétaires, bien que compatibles avec la spécification EMV, définissent toutefois un certain nombre de restrictions et de choix d'implémentations propres. La somme de ces spécificités (pouvant aller jusqu'à la suppression de certaines commandes) peut conduire à des protocoles sensiblement différents.

Afin de réaliser la surveillance et l'audit de cible selon le mécanisme présenté dans la section , nous avons développé des implémentations de référence dans le langage Python de chacune des deux spécifications EMV propriétaires les plus répandues.

A partir des types de données élémentaires du fuzzer, nous avons construit l'ensemble des commandes qui composent ces protocoles. Les spécifications des protocoles propriétaires étant confidentielles, le 1 présente à titre d'exemple les

champs obligatoires qui composent la commande *GENERATE APPLICATION CRYPTOGRAM* selon la spécification EMV CPA [28], leur modélisation à l'aide des types élémentaires fournis par le fuzzer et le nombre de cas de tests résultants.

Primitive	Val. Défaut	Type	Cas de tests
Header	80AE	Static	0
Trans. type	4000	Group	[*2]
Lc	23	Size	-
Amount	000000000010	Bit_field (BCD)	141
Amount other	000000000000	Bit_field (BCD)	141
Term country code	FFFF	Bit_field (hexa)	141
TVR	0000000000	Bit_field (hexa)	141
Currency code	0978	Bit_field (hexa)	141
Transaction year	09	Bit_field (BCD)	112
Transaction month	10	Bit_field (BCD)	112
Transaction day	18	Bit_field (BCD)	112
Transaction type	00	Group	9
Unpredictable Num.	11223344	Bit_field (hexa)	141
Terminal type	22	Group	16
CVR1	00	Bit_field (hexa)	112
CVR2	00	Bit_field (hexa)	112
CVR3	00	Bit_field (hexa)	112
Total test cases			3086

Table 1. Commande *GENERATE APPLICATION CRYPTOGRAM*.

Nous avons de la même façon modélisé les principales commandes des protocoles testés, puis nous les avons liées entre elles afin de modéliser le graphe d'état du protocole EMV (figure 5).

Tous les chemins du graphe sont explorés durant le test. Afin de tester une commande, chaque cas de test est préfixé par les commandes précédentes dans le modèle (pour les différents chemins possibles), avec les valeurs de primitives par défaut. Cette approche permet de tester le protocole en profondeur, car le comportement de l'application peut être différent selon l'enchaînement de commandes menant à un cas de test particulier.

Au total, un fuzzing complet du protocole selon ce modèle représente approximativement 146000 cas de tests.

Nous avons soumis les cartes à notre disposition à ces cas de test, ce qui nous a permis de mettre en lumière un certain nombre de failles fonctionnelles et/ou sécuritaires dans les applications testées.

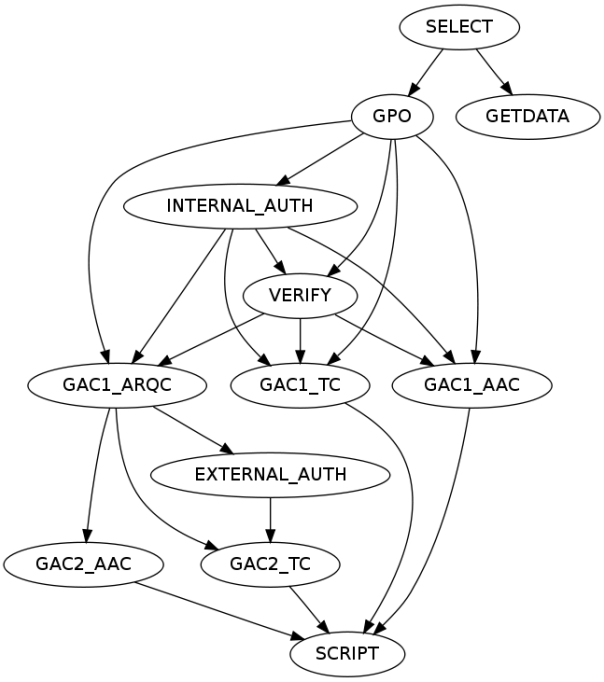

Figure 5. Modèle du protocole EMV.

4.2 Retours d'expérience

Nous avons testé 10 cartes à puce implémentant des applications M/Chip et VIS en les soumettant au test par fuzzing. Parmi les 10 cartes soumises au test, 6 ont révélé des défauts, dont 4 écarts fonctionnels et 1 faille sécuritaire présente sur deux produits. Le tableau 2 synthétise les résultats obtenus [2].

Détail des résultats obtenus Carte 1.1 et 1.2. Ces deux cartes embarquent deux versions successives de la même application. Nous avons identifié la même faille sur les deux versions. Cette faille concerne les compteurs de transactions « offline » qui permettent de limiter le nombre de transactions sans connexion vers l'émetteur. Dans les deux implémentations testées, une combinaison de valeurs pour les champs de la commande *GENERATE APPLICATION CRYPTOGRAM* entraîne la remise à

2. Les développeurs ont été informés des résultats obtenus sur leurs produits. Les applications concernées ont été patchées afin de corriger les failles découvertes. Pour des raisons évidentes de confidentialité, le nom réel des produits testés n'est pas fourni.

Identification	Type	Objet de la faille
Carte 1.1	Sécuritaire	Compteurs de transactions « offline »
Carte 1.2	Sécuritaire	Compteurs de transactions « offline »
Carte 2	Fonctionnel	Réponse de la commande GENERATE APPLICATION CRYPTOGRAM
Carte 3	Fonctionnel	Réponse de la commande GENERATE APPLICATION CRYPTOGRAM
Carte 4	Fonctionnel	Compteurs de transactions « offline »
Carte 5	Fonctionnel	Compteurs de transactions « offline »

Table 2. Résultats obtenus par fuzzing du protocole EMV.

zéro des compteurs. Cette faille est considérée comme sécuritaire car elle permet d'effecteur un nombre illimité de transactions sans contacter l'émetteur.

Carte 2 et 3. Dans la commande *GENERATE APPLICATION CRYPTOGRAM*, un élément de la commande envoyée par le terminal détermine le mode de calcul d'un des champs de la réponse.

Dans l'application de la carte 2, ce champ est calculé de manière identique quel que soit la commande envoyée par le terminal. Cet écart vis-à-vis de la spécification n'a pas d'impact sécuritaire, mais a un impact fonctionnel important car le champ considéré est impliqué dans un mécanisme d'authentification. Dans le cas où la valeur renvoyée par la carte est erronée, cette authentification échouera de manière certaine.

Dans l'application de la carte 3, le calcul du champ est conditionné par les commandes précédentes, ce qui n'est pas conforme aux spécifications. Comme pour la carte 2, le champ considéré est impliqué dans un mécanisme d'authentification, mais l'impact est moins important. En effet, la valeur erronée n'apparaît que si les commandes de la transaction sont incohérentes entre elles, ce qui ne peut pas se produire dans un cas nominal.

Carte 4 et 5. Les écarts de spécification identifiés dans ces deux applications concerne l'incrémentation de compteurs « offline » dans des cas non conformes à la spécification. Ces écarts n'ont pas d'impact sécuritaire. En effet, dans les deux cas, l'incrémentation des compteurs constitue une gestion du risque plus stricte que celle proposée par la spécification.

5 Travaux futurs

Les fuzzers que nous avons développés à l'aide de notre framework de fuzzing nous ont permis de découvrir un certain nombre de failles dans les produits EMV testés. Nous envisageons d'étendre nos tests à d'autres protocoles dédiés aux cartes à puce.

Comme les spécifications EMV présentées dans cet article, les spécifications ICAO MRTD (e-passeport) [25], MONEO (micro-paiement) [26] et IAS (e-administration) [27] sont volumineuses et complexes. Les mêmes risques d'introduire des failles sécuritaires ou fonctionnelles dans les implémentations de ces spécifications existent donc.

Afin d'identifier ces failles, nous comptons développer des fuzzers pour ces spécifications en exploitant les fonctionnalités offertes par notre framework de fuzzing. Le développement de ces fuzzers nécessite dans un premier temps la réalisation des implémentations de référence de ces spécifications nécessaire à la surveillance de la cible. Ensuite, le modèle des protocoles sera implémenté pour permettre la génération des données de test.

6 Conclusion

Les puces électroniques offrent aujourd'hui une puissance de calcul considérable et embarquent des mécanismes de sécurité matériels avancés. Cependant, cette sécurité potentielle offerte par la plateforme matérielle peut être mise en échec par des failles d'implémentation au niveau applicatif.

Afin de vérifier que les applications embarquées sur les cartes à puce sont exemptes de failles sécuritaires et fonctionnelles, nous avons mis en œuvre une technique de test par fuzzing. Cette technique de test en boite noire, traditionnellement utilisée pour la recherche de vulnérabilités dans les implémentations de protocoles réseaux, s'avère particulièrement adaptée au domaine des cartes à puce.

Nous avons développé un framework de fuzzing pour cartes à puce qui permet d'explorer en profondeur les protocoles dédiés aux cartes à puce. A l'aide de ce framework, nous avons implémenté deux fuzzers pour les protocoles de paiement sécurisé les plus répandus.

Les résultats obtenus par fuzzing sur notre échantillon de cartes à puce montrent qu'en ce qui concerne les failles sécuritaires, les techniques de tests par fuzzing ne sont pas fondamentalement plus efficaces que les techniques de tests classiques. En effet, une seule vulnérabilité réellement sécuritaire a été identifiée par fuzzing. Ceci s'explique par le fait que les applications testées sont développées avec une approche résolument sécuritaire et offrent globalement un haut niveau de sécurité. Cependant, le fuzzing a révélé un certain nombre d'écarts par rapport aux spécifications dans les implémentations testées, ce qui s'explique par la complexité des protocoles dédiés aux cartes à puce. Ces erreurs n'ont pas d'impact sécuritaire, mais ont parfois un impact fonctionnel important, et n'ont pas été détectées durant la campagne de tests fonctionnels menée par les développeurs. Le test par fuzzing pour les cartes à puces

apparaît donc complémentaire aux approches de test classiques. Cette technique de test pourrait être intégrée de manière systématique au processus de développement durant la phase de validation afin de détecter et corriger au plus tôt les failles sécuritaires et fonctionnelles des applications embarquées sur cartes à puces.

Références

1. IMS Research, *The World Market for Smart Cards and Smart Card Ics*, janvier 2010.
2. EMVCo, *About EMV*, novembre 2010, http://www.emvco.com/about_emv.aspx
3. EMV – *Integrated Circuit Card Specifications for Payment Systems, Book 1 : Application Independent ICC to Terminal Interface Requirements*, Version 4.2 ed., EMVCo, LLC, Juin 2008.
4. EMV – *Integrated Circuit Card Specifications for Payment Systems, Book 2 : Security and Key Management*, Version 4.2 ed., EMVCo, LLC, Juin 2008.
5. EMV – *Integrated Circuit Card Specifications for Payment Systems, Book 3 : Application Specification*, Version 4.2 ed., EMVCo, LLC, Juin 2008.
6. EMV – *Integrated Circuit Card Specifications for Payment Systems, Book 4 : Cardholder, Attendant, and Acquirer Interface Requirements*, Version 4.2 ed., EMVCo, LLC, June 2008.
7. ISO – *Information technology – Identification cards – Integrated circuit(s) cards with contacts – Part III : Electronic signals and transmission protocols*, 1997. Reference Number : ISO/IEC 7816-3.
8. ISO – *Information technology – Identification cards – Integrated circuit(s) cards with contacts – Part IV : interindustry commands for interchange*, 1995. Reference Number : ISO/IEC 7816-4.
9. VISA, *Visa Integrated Circuit Card Specification (VIS)*, Version 1.5, May 2009
10. Godefroid, P., Levin, M.Y., Molnar, D., *Automated Whitebox Fuzz Testing*, NDSS, 2008
11. Ganesh, V., Leek, T., Rinard., M., *Taint-based directed whitebox fuzzing*, IEEE 31st International Conference on Software Engineering, 2009
12. Clarke, T., *Fuzzing for software vulnerability discovery*, Royal Holloway, University of London Technical Report, Février 2009
13. Aitel, D., *The Advantages of Block-Based Protocol Analysis for Security Testing*, Immunity Inc., Février 2002
14. Deja Vu Security, *Peach Fuzzing Platform*, http://www.peachfuzzer.com
15. Amini, P., Protnoy, A., Sulley Fuzzing Framework,http://code.google.com/p/sulley
16. Butti , L., *Sécurité des Architectures de Convergence Fixe-Mobile, Symposuim sur la Sécurité des Technologies de l'Information et de la Communication* (SSTIC), 2009
17. *Python Programming Langage*, http://www.python.org
18. Mulliner, C., Miller, C., *Injecting SMS Messages into Smart Phones for Security Analysis*, 3rd USENIX conference on Offensive Technologies (WOOT), 2009
19. MISCMAG 39, Dossier fuzzing, http://ed-diamond.com/feuille_misc39/index.html
20. Bratus, S., Hansen, A., Shubina, A., *LZfuzz : a fast compression-based fuzzer for poorly documented protocols*, Dartmouth Computer Science Technical Report
21. MasterCard, *M/Chip 4 Card Application Specifications for Debit and Credit - Part A*, Version 1.1, Septembre 2006
22. Molnar, D., Li, X. C., Wagner, D., *Dynamic test generation to find integer bugs in x86 binary linux programs*, 18th USENIX Security Symp. USENIX Association, aout 2009

23. Sutton, M., Greene, A., *The Art of File Format Fuzzing*, BlackHat, septembre 2004
24. Zimmer, D., COMRaider, iDefense Labs, http://labs.idefense.com/fuzzing.php
25. ICAO, *Machine Readable Travel Document – PKI for Machine Readable Travel Documents offering ICC Read-Only Access*, Version 1.1, Janvier 2004
26. MONEO, *Electronic Purse – Card Specification – PME*, version 2.5.2, Décembre 2001
27. GIXEL, *Plateforme commune pour l' eADMINISTRATION – Spécification Technique*, Révision : 1.01, Novembre 2006
28. EMV – *Integrated Circuit Card Specifications for Payment Systems, Common Payment Application Specification*, Version 1.0 , EMVCo, LLC, Décembre 2005
29. PC/SC workgroup, *PC/SC workgroup specifications*, version 2.01.9, http://www.pcscworkgroup.com/specifications/specdownload.php