Évaluation de l'injection de code malicieux dans une Java Card

Julien Iguchi-Cartigny, Jean-Louis Lanet
{julien.cartigny,jean-louis.lanet}(@)unilim.fr

XLIM/DMI/SSD 83 rue d'Isle
87000 Limoges

Résumé Nous avons évalué différentes cartes à puce en vérifiant leur résistance à l'attaque EMAN qui porte sur une faiblesse d'un composant de la machine virtuelle Java Card : le *firewall*. Les hypothèses de cette attaque sont : l'absence d'un vérificateur de byte-code à l'intérieur de la carte (ce qui est le cas dans la plupart des cartes évaluées), et l'autorisation du chargement a posteriori de code dans la carte (nous possédons les clés de chargement). Ce papier présente les résultats de l'étude comparative des contre-mesures intégrées dans les cartes. Nous montrons que certaines cartes à puce sont particulièrement sensibles à cette attaque et qu'il est possible de divulguer du code auquel nous n'avons normalement pas accès.

Mots-clés: carte à puce, attaque logique, *firewall*, Java Card

1 Introduction

Une carte à puce peut être vue comme un support sécurisé de manipulation de données sensibles, car elle stocke des données d'une façon sûre et elle les manipule d'une manière fiable pendant la durée des transactions. Pour pouvoir résister contre des attaques externes, une carte à puce dispose de plusieurs niveaux de défense. Le premier est lié au matériel. Une attaque physique est difficile à mener car le microprocesseur et ses capteurs physiques sont enrobés dans une résine. De plus, tous les composants sont sur le même silicium (ce qui rend difficile la pose de sonde sur le bus interne). Le logiciel est la deuxième barrière de sécurité : le système d'exploitation et les applications sont conçus pour ne jamais renvoyer des informations sensibles sans s'assurer que l'opération soit autorisée. Une carte à puce de type Java Card est une plate-forme ouverte c'est-à-dire capable de charger et d'exécuter de nouvelles applications après la phase de délivrance (*post-issuance*). Il existe plusieurs mécanismes garantissant l'isolement des applications. Ainsi, des applications de différents fournisseurs peuvent s'exécuter dans la même carte à puce sans possibilité d'accès à l'espace de travail d'un autre fournisseur. Puisqu'il est possible de charger de nouvelles applications, cet isolement s'appuie sur les propriétés du système de typage fort de Java et la vérification du respect des règles par le vérifieur de type. Il est ainsi possible de garantir

qu'une application chargée dans la carte n'est pas hostile aux autres applications. En outre, le pare-feu (*firewall*) implémenté dans les Java Cards vérifie lors de l'exécution que les accès respectent bien ces règles imposant l'isolement entre les applications. Jusqu'à présent, il était sûr de présumer que le *firewall* était efficace pour éviter un comportement malveillant de la part d'une ou plusieurs applications. Dans cette étude, nous montrons qu'un attaquant peut produire des applications malveillantes qui sont non détectées par le *firewall* et qui modifient d'autres applications, même si elles ne sont pas dans le même contexte de sécurité. Le *firewall* étant le seul mécanisme de sécurité obligatoire intégré dans une Java Card, ce travail pointe un problème de sécurité évident. Dans la prochaine section, nous décrivons les différents composants impliqués dans la sécurité de la plate-forme Java Card. La section 3 présente l'état de l'art des attaques logiques contre les cartes à puce. Nous présentons dans la section 4 une méthode pour mettre en application notre attaque avec l'introduction d'un cheval de Troie dans la carte. La section 5 présente les premiers résultats de notre attaque et les contre-mesures détectées. La section 6 présente nos conclusions.

2 Java Card

2.1 Présentation

Java Card est l'adaptation de la technologie Java pour les cartes à puce. Les applets sont développées dans le langage Java, elles sont ensuite transformées afin de satisfaire les contraintes de mémoire de la carte puis sont chargées dans les cartes. Une fois installée sur la plate-forme et sélectionnée, le byte code de l'applet est interprété par la machine virtuelle embarquée (Java Card Virtual Machine, JVCM), laquelle est définie par les spécifications de Sun [2]. En raison des contraintes de ressource, la JVCM doit être séparée en deux parties :

- en dehors de la carte : le vérifieur de byte code (BCV) et le convertisseur. Ils sont exécutés pour produire un fichier au format CAP,
- à l'intérieur de la carte : l'interpréteur, l'API et l'environnement d'exécution de carte de Java (JCRE).

Le vérifieur réalise une analyse statique de code par interprétation abstraite sur les composants formant le paquetage. Puis, le convertisseur transforme ce fichier en format plus approprié pour la carte : le format CAP (contient une représentation compacte d'un ou plusieurs *class file*). L'étape suivante est le chargement, l'édition de lien et le stockage des classes dans la mémoire de la carte par le chargeur (*Class Loader*). Pendant le processus de chargement, les éléments du fichier CAP sont chargés dans la carte séquentiellement, composant par composant par le chargeur qui stocke

son contenu dans la mémoire persistante. Après cette phase de chargement et l'édition de lien, le paquetage est prêt à être exécuté par la JCVM.

2.2 Sécurité

La plate-forme Java Card est un environnement de cartes multi-applicatives dans laquelle les données sensibles d'une applet doivent être protégées contre l'accès malveillant pouvant être réalisé par d'autres applets [5]. Pour imposer la ségrégation entre applets, la technologie Java dans les éditions autres que Java Card, utilise la technique de vérification, le chargeur de classe, le contrôleur d'accès et le gestionnaire de sécurité. Une carte à puce n'utilise pas toutes ces techniques. D'abord, le vérifieur de type est placé en dehors de la carte, ceci étant dû aux contraintes de mémoire. Ce qui implique que le chargement d'applet doit être fait dans un environnement sécurisé ou en s'assurant de l'origine du code. C'est le rôle de la couche logicielle Global Platform qui utilise des protocoles d'authentification avant chargement afin d'autoriser ou non le chargement d'applet. Ensuite il n'existe qu'un seul chargeur de classe et il est impossible de redéfinir son comportement. Enfin, le gestionnaire de sécurité et le contrôleur d'accès sont remplacés par le *firewall* dans Java Card. La vérification de byte code est un composant crucial de la sécurité dans le modèle d'isolation de Java. Une erreur dans l'implémentation du vérifieur peut autoriser une applet mal typée d'être acceptée et potentiellement mener à une défaillance. En même temps, la vérification de byte code est un processus complexe et coûteux en termes de CPU et de mémoire. Pour ces raisons, beaucoup de cartes ne mettent pas en application un tel composant à l'intérieur. La sécurité du système dépend alors de l'organisation qui signe le code, laquelle doit s'assurer que le code est bien typé. L'isolement entre les applets à l'intérieur de la carte est imposé par le *firewall* de Java Card qui n'autorisera que les interactions entre les applets appartenant au même contexte. À chaque paquetage Java Card est associé un identifiant unique (AID) et toute applet de ce paquetage reçoit un identifiant associé à l'identifiant du paquetage. D'une manière plus précise, chaque objet (tableau ou instance de classe) sur la carte appartient au contexte de l'applet qui l'a instancié, c'est-à-dire l'applet qui était active lorsque l'objet a été créé. Une applet a le droit d'accéder à ses objets (et à chaque objet situé dans le même paquetage), mais le *firewall* vérifie toujours qu'une applet n'essaye pas d'accéder illégalement à un objet n'appartenant pas à son contexte. Ainsi, le firewall isole les contextes de telle manière qu'une méthode étant exécutée dans un contexte ne puisse accéder à aucun champ ou méthodes d'objets appartenant à un autre contexte. Nous disposons ainsi d'un mécanisme efficace pour garantir la ségrégation entre applications. Néanmoins, il faut noter qu'il existe un mécanisme spécifique permettant malgré tout l'échange de données entre applications

de contextes différents, il s'agit du protocole de partage SIO (Sharing Interface Object) qui spécifie comment les références sur des objets partagés peuvent être obtenues et quels sont les contrôles à effectuer à cet instant.

3 État de l'art des attaques sur carte à puce

3.1 Attaques physiques

Une première classe d'attaques physiques est l'attaque par canal auxiliaire [3]. Ces attaques non envahissantes consistent à observer un effet physique lié à un calcul (synchronisation, données échangées sur les canaux d'entrée-sortie, énergie consommée, bruit électromagnétique, etc.) pour découvrir de l'information comme une clef secrète utilisée dans une fonction cryptographique. Par exemple, la SPA (*simple power analysis*) et la DPA (*differential power analysis*) visent à exploiter l'information obtenue par des variations caractéristiques de la consommation d'énergie des composants électroniques. La seconde classe d'attaque est l'injection de faute. Cette attaque consiste à changer le comportement d'un composant en le perturbant afin de créer une erreur exploitable [4]. De tels défauts peuvent être induits par différents moyens, y compris des fautes transitoires de courte durée (altération rapide de l'alimentation), l'ajout d'énergie par rayonnement laser, etc. L'attaque vise à rendre des opérations cryptographiques moins sûres, obtenir plus facilement des clefs, ou à modifier les flots de contrôle du programme en altérant la mémoire, les informations transitant sur les bus ou les registres du processeur.

3.2 Attaques logiques

Comme expliqué dans [1], les attaques logiques consistent à exécuter des applications malveillantes, par exemple des applications pour lesquelles les instructions ne respectent pas les règles de typage de Java ou les règles de construction des fichiers d'entrée, ou encore utilisent des imprécisions d'une ou plusieurs des spécifications Java Card. Afin qu'un attaquant puisse charger son code malveillant, les hypothèses les plus communément admises pour les attaques logiques sont :

- la carte autorise le chargement *post-issuance*,
- l'attaquant possède les clefs de chargement,
- la carte ne possède pas de vérificateur de byte code intégré.

E. Poll et W. Mostowski [7] proposent plusieurs attaques par confusion de type. L'objectif est de transformer un objet d'un type donné pour lequel un nombre limité d'opérations est possible en un objet pour lequel d'autres opérations deviennent éligibles. Par exemple, transformer un tableau d'octets en un tableau d'entiers permet

de pouvoir lire deux fois plus de données et potentiellement des données sortant du domaine de l'applet. Une autre approche (d'abord proposée par Witteman [9] puis par Vertanen [8]) est de faire traiter à la machine virtuelle un objet en tant que tableau. Si les attributs du descripteur d'objet sont correctement renseignés et situés aux mêmes emplacements mémoire alors il devient possible de modifier la taille du tableau en tant qu'attribut légitime d'un objet. À partir de là, il est possible de l'utiliser en tant que tableau ayant une taille égale à la mémoire de la carte et il est possible de lire/écrire partout en mémoire à partir de l'adresse à laquelle est stocké l'objet. Les auteurs exploitent l'attaque précédente pour traiter une référence vers un objet en tant que short. De par ce transtypage, il est possible de lire et modifier des références existantes, chose théoriquement impossible sur une Java Card. Ils ont ainsi proposé plusieurs exploitations de cette méthode. Par exemple, il est possible d'échanger la référence de deux objets même si ceux-ci ont des types incompatibles. Un attaquant peut également manipuler l'AID associé à un objet et rendre invalide ainsi une applet dûment référencée par le système. De même, il devient possible de fabriquer des références et lire une partie de la mémoire. Néanmoins les auteurs expliquent qu'un certain nombre de contre mesures sont implémentées dans les cartes rendant cette attaque moins sensible. Hyppönen dans sa thèse [6] suggère une autre approche. Il s'agit d'exploiter une faiblesse de l'utilisation du *firewall* avec les instructions manipulant des attributs globaux (`getstatic` et `putstatic`). L'instruction `getstatic` est employée pour lire le contenu d'un champ statique d'une classe. Les deux opérandes de cette instruction sont employés pour établir un index dans le constant pool. Pendant le chargement d'applet, le processus d'édition de lien remplacera les deux opérandes par une adresse dans la mémoire. L'idée de l'attaque est de supprimer l'information qui demande la résolution de cette référence de sorte que cette dernière ne soit pas modifiée par l'édition de lien. Ceci est rendu possible de par l'utilisation d'un composant recensant les champs devant être résolus lors du chargement. Cette attaque fonctionne (en l'absence d'un vérificateur de byte code) parce qu'aucun contrôle de contexte n'est fait par le *firewall* pendant l'accès au champ statique. Cependant, l'auteur n'a présenté aucun résultat expérimental ou même une implémentation de l'attaque. Dans ce document, nous montrons qu'une telle attaque est possible, et nous proposons une exécution très efficace qui nous permet de produire d'un code auto modifiable.

4 Réaliser un cheval de Troie dans une carte

Dans l'approche de Hyppönen nous ne pouvons au mieux que modifier le contenu d'une seule adresse. Afin de pouvoir exploiter réellement cette approche nous avons

développé un code auto modifiable nous permettant de lire et écrire n'importe où en mémoire de la carte. À partir de là, nous avons implémenté une fonction de recherche et de remplacement de motifs afin de remplacer des parties de code dans la carte. Afin de montrer la puissance de cette attaque, considérons le code suivant très souvent utilisé pour la vérification du PIN code dans une Java Card. Il est basé sur l'utilisation de l'API Java Card et de l'objet OwnerPin qui est une implémentation sécurisée (décrémentation du compteur de ratification avant la comparaison, etc) de l'objet modélisant un Pin code. Dans le fragment de code suivant, lorsque l'applet veut réaliser un débit, elle se prémunit en vérifiant si le Pin code a déjà été validé auparavant en utilisant la méthode isValidated () sur l'objet Pin code. Ce fragment de code est très classique dans son utilisation. Si l'utilisateur entre un mauvais code alors une exception est émise et la fonction est abandonnée.

```
public void debit (APDU apdu )
{
  ...
  if (!pin.isValidated())
    {
     ISOException.throwIt(SW_AUTH_FAILED)
    }
    // do safely something authorized
}
```

L'objectif de notre cheval de Troie est de rechercher dans le code de cette application (qui évidemment n'appartient pas au contexte de sécurité de l'attaquant et ne lui est même pas accessible en lecture) le byte code traitant l'exception. Par exemple, si notre attaque détecte en mémoire le code `11 69 85 8D 00 12` et si le propriétaire du code est l'applet ciblée alors il suffit de remplacer ce code par le motif suivant : `00 00 00 00 00 00`. Le byte code 00 étant celui qui code l'instruction NOP, le fragment de code de l'applet chargée devient :

```
public void debit (APDU apdu )
{
...
if (!pin.isValidated())
{ }
// do safely something authorized
}
```

L'intérêt de cette fonction de recherche et remplacement devient évident : il est possible de contourner les fonctions de sécurité de n'importe quelle applet chargée dans la carte. Si le cheval de Troie est capable de lire et écrire en mémoire il devient aussi possible de l'utiliser pour caractériser la représentation des objets pour la machine virtuelle embarquée. Il devient aussi possible d'obtenir le code d'implémentation des algorithmes cryptographiques selon l'espace mémoire où ils sont implémentés, lequel peut à son

tour générer de nouvelles attaques. Comme mentionnées plus haut, les hypothèses de base de notre attaque sont très classiques, nous disposons des droits de chargement ainsi que des clés pour le réaliser et les cartes ne disposent pas de vérifieurs de byte code. Nous sommes donc dans le contexte des cartes de développement (accessible sur n'importe quel webstore) et non des cartes de production. Mais nous verrons que notre travail peut potentiellement avoir un impact sur ces dernières.

4.1 Description de l'attaque, la phase de pre-link

La première étape consiste à obtenir une référence sur un tableau situé dans le contexte de sécurité de notre propre applet. Supposons le code java suivant :

```
public short getMyAddresstabByte(byte[] tab)
{
short dummyRef=(byte)0x55AA;
tab[0] = (byte)0xFF; // second instruction
return dummyRef;
}
```

Au niveau du byte code, nous voyons que la seconde instruction est un aload_1 donc on en déduit que la référence au tableau est placée au sommet de la pile après la seconde instruction. Si toutes les instructions suivantes sont remplacées par des NOP alors la fonction renverra à la place de dummyRef la référence au tableau passé en paramètre. Il ne reste plus qu'à renvoyer au terminal cette référence dans l'APDU de retour de la carte. Le programme pilotant l'applet obtient donc une référence valide d'un tableau situé à l'intérieur de la carte.

4.2 Une applet auto modifiable

Nous commençons par définir une variable de type tableau (appelée ci-après codeD) que nous allons exécuter. La définition est la suivante :

```
public byte[] codeD = {(byte)0x01, (byte)0x00,
                       (byte)0x7D, (byte)0x00,
                       (byte)0x00, (byte)0x78};
```

Si on considère ce tableau comme étant une fonction qui sera exécutée, alors cette dernière réalise la lecture d'un champ statique dont l'adresse pourra être fournie avant l'appel. Les deux premiers octets de ce tableau correspondent au descripteur de la méthode tel que défini par Sun. Ils seront donc interprétés comme suit :

```
//flags : 0
//max_stack : 1
//nargs : 0
```

```
//max_locals : 0

00 getstatic_s 0 0
03 sreturn
```

Avec une commande APDU nous pouvons modifier le quatrième et le cinquième octet de ce tableau/méthode qui sont donc les paramètres de l'instruction `getstatic` :

```
codeD[3]= apduBuf[ISO7816.OFFSET_CDATA];
codeD[4]= apduBuf[ISO7816.OFFSET_CDATA+1];
```

Il nous reste à définir une référence à une méthode fictive *functionToReplace()*, afin de générer le code d'invocation à cette méthode statique.

```
// function to replace
static public short functionToReplace()
{
return ad;
}
```

Nous pouvons dès lors écrire une boucle pour cheval de Troie réalisant la fonction chercher-remplacer :

```
For (i=0...){
 // to generate a ref to be replaced later
Util.setShort(searchBuf,k,functionToReplace());
codeDump[4]++; //increment low address
if (codeDump[4] == (byte)0x00)
 {
  codeDump[3]++; // increment high address
 }
// search and replace the pattern in searchBuf
}
```

La méthode décrite par Hyppönen consiste à leurrer l'édition de lien embarquée en modifiant le composant servant à accélérer cette phase du chargement. Dans l'exemple de la figure 1, la méthode contient une référence (0002) à un champ du constant pool. Lorsque le constant pool est chargé dans la carte, une adresse physique lui est assignée (ici pour l'entrée n° 2 : 0x8805) et ce pour chaque entrée. Ensuite, le composant Reference Location est chargé dans la carte (en 1 sur la figure), indiquant à l'édition de lien que dans une des méthodes à l'offset 000F (l'offset est global à l'ensemble des méthodes) il trouvera une entrée à résoudre. Le composant Method est ensuite chargé dans la carte (2) et le linker remplace à l'offset 000F la valeur de l'entrée 2 par son adresse physique en mémoire (3) assignée précédemment. Nous utilisons alors le même phénomène que celui pointé par Hyppönen mais cette fois sur le byte code `invokestatic`, en remplaçant l'entrée dans le constant pool de la méthode `functionToReplace()` par l'adresse obtenue précédemment

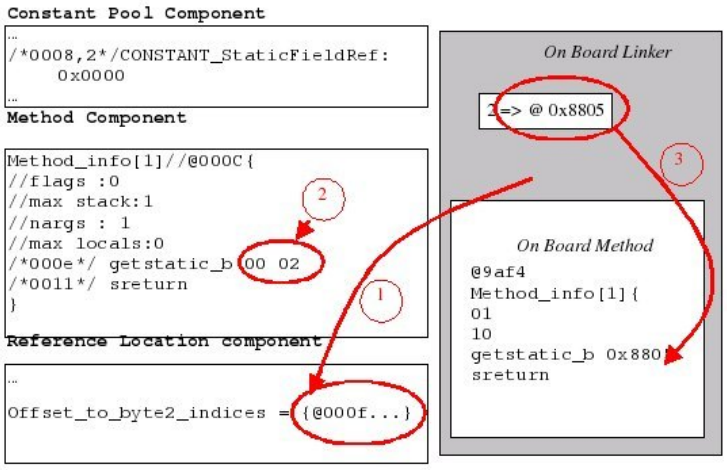

Fig. 1. Édition de lien lors du chargement dans la carte

pour notre tableau `codeD`. À l'offset 0x014e dans le composant `Method`, on retrouve l'instruction invokstatic concernant l'appel à la méthode `fonctionToReplace()`. Nous pouvons voir que l'opérande est un index dans le constant pool et que l'offset est référencé dans le composant `Reference Location`. Il faut d'abord éditer le fichier, modifier la valeur contenue à l'adresse `0x014e` par la valeur de son successeur dans le composant `Reference Location`. Ainsi l'éditeur de lien va résoudre deux fois l'adresse symbolique suivante, laquelle correspond à la méthode `setShort()`. Dans une seconde étape, nous remplaçons les opérandes de l'invokestatic par l'adresse du tableau `codeD` ; il est ensuite possible de charger et installer l'applet. En fait, il est nécessaire de travailler en deux étapes : la première récupère l'adresse du tableau et la seconde réalise la vraie installation. Dès lors, il est possible à l'aide d'une commande APDU envoyée à cette applet de choisir la zone à lire et écrire et de réaliser la fonction chercher-remplacer. Dès que le premier octet à chercher est détecté, le contenu de la mémoire est stocké dans le tableau `earchBuf[k]`. Si le byte code suivant n'est pas élément du motif recherché, le programme réinitialise la valeur de k et continue la recherche, sinon une APDU de réponse est envoyée indiquant la découverte du motif et son remplacement. En fait le cheval de Troie repose sur une connaissance de la représentation interne des structures de données utilisées pour chaque type de carte. Évidemment une telle information n'est pas disponible publiquement et il nous a fallu connaître comment chaque *Class Loader* transforme le fichier d'origine dans sa propre représentation. En fait le fichier CAP est très proche de l'optimum en termes

de représentation exécutable et peu de cartes font des modifications fondamentales. Ayant acquis cette connaissance, notre méthode de recherche-remplacement utilise ces informations pour affiner la recherche. Par exemple, la connaissance du lieu de stockage de l'AID de la classe ayant généré l'applet nous permet de cibler notre remplacement uniquement à l'applet recherchée. La difficulté la plus grande consiste à modifier de façon cohérente les différents composants du fichier CAP. Lors de ces travaux nous ne disposions pas d'un tel outil permettant l'édition, la modification et la reconstitution du fichier CAP. Nous disposons désormais d'un tel outil sous la forme d'une bibliothèque Java pouvant être intégrée dans un outil de génération de code mutationnel.

5 Évaluation de l'attaque

Nous avons conduit cette attaque sur plusieurs types de Java Card. Certaines comportaient des contre-mesures rendant inefficace notre attaque, d'autres nous ont laissés partiellement l'exécuter en contournant certaines contre-mesures et d'autres cartes ne résistent pas à cette attaque. Il est donc possible sur certaines cartes d'invalider à distance des contrôles de sécurité à l'aide de notre cheval de Troie. Les cartes évaluées sont toutes disponibles publiquement via internet. Nous avons évalué neuf cartes de trois fournisseurs (a, b et c) distincts. Nous identifierons les cartes en utilisant une référence à l'un des fournisseurs associée à la version de la spécification utilisée. Au moment de cette étude, aucune carte de type Java Card 3.0 n'était disponible et les plus récentes versions étaient des Java Card 2.2.

- fournisseur A, cartes a-21a, a-21b, a-22a and a-22b. La carte a-22a est une carte de type USIM pour la téléphonie de troisième génération, la a-21b est une extension de la a-21a supportant l'algorithme RSA, et la a-22b est une carte possédant deux interfaces avec ou sans contact.
- fournisseur B, cartes b-21a, b-22a, b22b. La b-21 supporte l'algorithme RSA, et la b-22b est elle aussi une carte à double interface.
- fournisseur C, carte c-22a, c22b. La première est une carte avec deux interfaces la seconde supporte quant à elle l'algorithme RSA.

La table suivante rappelle les différentes caractéristiques des cartes évaluées.

Nous donnons ci-après la représentation interne d'un objet appartenant à l'une des cartes attaquées. La représentation des tableaux statiques dans la mémoire est le suivant : le premier octet décrit le type Java du tableau (81 pour le type octet, 83 pour un short et 85 pour des booléens). Les deux octets suivants décrivent la taille, puis les quatre octets suivants représentent deux pointeurs, l'un dans la zone rom et l'autre dans la zone eeprom. Puis suivent les données du tableau.

Tab. 1. Définition des cartes utilisées dans cette étude

Référence	Java Card	GP	Caractéristiques
a-21a	2.1.1	2.01	
a-21b	2.1.1	2.0.1	Same as a-21a plus RSA
a-22a	2.2	2.1	64k Eeprom
a-22b	2.1.1	2.0.1	32k Eeprom, RSA
b-21a	2.1.1	2.1.2	16k Eeprom, RSA
b-22a	2.1.1	2.0.1	16k Eeprom, hW DES
b-22b	2.1.1	2.1.1	
c-22a	2.1.1	2.0.1	RSA
c-22b	2.2	2.1.1	64 k Eeprom, dual interface, RSA

5.1 Extraction de fragment de code

Le premier effet de cette attaque est de pouvoir parcourir la mémoire afin de détecter des motifs connus. Dès que le motif est détecté, il est possible de modifier (sur certaines cartes) le contenu du code de l'applet. Dans la pratique, certaines zones mémoire ne sont pas accessibles en lecture comme la zone rom qui correspond au code de la machine virtuelle et des API. Notre intérêt actuel concerne l'accès au contenu des applets et comme ces dernières sont stockées dans la zone eeprom, ceci ne représente pas un inconvénient majeur. Il l'est si nous cherchons à comprendre l'implémentation des algorithmes cryptographiques qui eux sont dans la zone rom, et un autre mécanisme sera nécessaire tel que nous le montrons plus loin. Une fois que nous obtenons une référence valide sur une instance d'un objet, il devient facile de se déplacer dans la hiérarchie de classe, comme indiqué sur la figure 1. Dans l'exemple suivant, nous obtenons une référence sur un objet stocké à l'adresse 0x8820. Les quatre premiers octets sont des éléments du descripteur d'objet, les deux suivants pointent l'adresse de la classe de cet objet.

```
@8820 .reference = {00 02 41 10 85 d2...}
```

Il devient donc possible d'analyser la structure de la classe. Le descripteur de classe est construit d'une manière très proche (pour la carte étudiée) du format du CAP file. Il est possible d'analyser facilement son contenu. On s'aperçoit que, dix octets après le début de la référence, on obtient une table des méthodes. On peut remarquer que la seconde méthode est stockée dans la zone rom, elle correspond à la méthode d'initialisation de la super classe de l'objet courant et donc d'applet.

```
@85d2 .class {
flags =  0
interface_count = 0
super_class_ref : 6d 91
```

```
declared_instance_size     : 01
first_reference_index      :00
reference_count            :01
public_method_table_base   :04
public_method_table_count  :08
unknown tag :00 00
@85DC public_methods = {@8689, @6ed9, @8685, @868c,  @85ed,
                        @85f8, @8604, @860c }
```

En observant la méthode stockée à l'adresse 0x85ed, il est possible de retrouver sa suite d'instructions après le descripteur de méthode, lequel est représenté par deux octets dans cette carte. Le descripteur correspond à la définition donnée dans le CAP file.

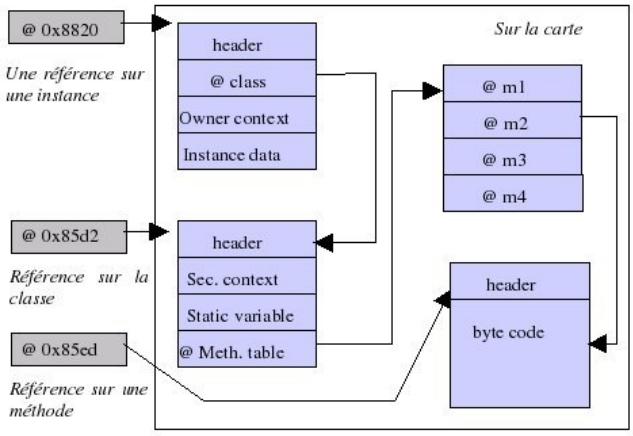

Fig. 2. Navigation dans la structure objet interne de la carte

```
@85ed method_info[4]  = {flags   : 0, max_stack : 3,
nargs : 2, max_locals: 1
    /*0003*/        bspush          -86
    /*0005*/        sstore_2
    /*0006*/        aload_1
...
}
```

Il est aussi possible de lire la zone de mémoire vive avec la même technique. Ainsi nous avons découvert la référence de l'instance de la classe APDU : 0x1D2. À cette adresse est stockée la structure suivante 00 04 29 FF 6F 0E. Cette structure représente l'instance de la classe APDU et donc nous pouvons déduire qu'à l'adresse 0x6F0E

est stockée la classe APDU dans la zone rom. Le buffer de l'APDU est situé dans la mémoire ram juste après le descripteur et il est possible de retrouver le contenu du buffer envoyé précédemment. La pile Java est située après le descripteur de l'instance d'APDU, on y retrouve l'empilement de la référence de l'APDU. Nous n'avons pas essayé d'exploiter l'information contenue dans la pile d'appel Java pour mener des attaques. L'analyse de la zone mémoire fait apparaître ce qui nous semble être la pile C. En effet, nous trouvons à l'extrémité de la zone mémoire des données qui semblent être des adresses mémoires pointant sur la zone rom. Sur ce composant qui est un microcontrôleur de type P8WE5017, soit une base de 8051, il semble qu'il n'y ait pas de restriction sur l'exécution de code en eeprom. Il devient alors (relativement) facile, puisque nous avons la possibilité d'écrire dans cette zone, de détourner l'avant-dernier retour vers une zone exécutable en eeprom qui contienne du code natif. Cette possibilité nous offre l'opportunité de prendre le contrôle du processeur et d'accéder à la zone rom. Nous rappelons que notre attaque EMAN d'origine agit au niveau de la machine virtuelle Java et donc nous ne pouvons accéder à la zone rom contenant le code programme natif. Il devient alors possible de reverser le code rom, de l'analyser et de chercher une faille pouvant être utilisée sur une carte de production. Cette possibilité reste évidemment à démontrer et nous n'avons pas encore eu le temps de l'implémenter.

5.2 Analyse du plan mémoire

Une des difficultés rencontrées avec l'analyse de la mémoire est la difficulté de retrouver des objets dans la densité des informations disponibles. Il est difficile de séparer ce qui correspond à des données, des structures d'objet et du bruit. S'il a été possible sur une carte d'extraire une partie de ces informations, l'automatisation devenait nécessaire pour analyser d'autres cartes de manière systématique. Nous avons donc conçu un outil permettant à partir d'un fichier contenant des données binaires brutes de définir des zones dans lesquelles la probabilité d'avoir des structures de classe était importante, de trouver des tables d'adresse et des objets spécifiques comme des clés d'authentification. Certaines cartes étaient plus difficiles à analyser car elles implémentent des gestionnaires mémoire rendant l'écriture des zones plus difficile. Notre outil utilise plusieurs heuristiques pour définir si la zone en cours d'analyse est une zone contenant une structure d'objet. La première heuristique est l'indice de coïncidence IC (1), utilisé habituellement pour reconnaître un langage. Il est basé sur la probabilité que certains caractères ont une probabilité plus élevée d'apparaître que d'autre. Par exemple cet index est de 0.0778 pour le français et 0.0667 pour l'anglais. Nous avons donc cherché l'indice de coïncidence pour le langage

Java Card (byte code). Après une phase d'apprentissage, nous avons conclu que ce langage avait un indice compris entre 0.009 et 0.025.

$$IC = \sum_{k=0}^{k=0xFF} n_i \frac{(n_i - 1)}{n(n-1)}$$

n_i est le nombre courant d'occurrence du caractère k

n est le nombre total de caractères dans le code

k est le caractère courant (octet)

La seconde heuristique utilisée concerne la probabilité d'occurrence d'une instruction illégale. Les instructions valides sont codées entre 0x00 et 0xb8, le reste est non utilisé. De plus certaines cartes n'implémentent pas les entiers et toutes les instructions y faisant référence deviennent illégales. Cependant ces valeurs peuvent représenter des opérandes, des constantes, etc. Après plusieurs essais nous avons défini un pourcentage de 2,5% au-dessus duquel la présence d'instructions illégales permet de garantir que la zone ne comprend pas des instructions Java. La dernière heuristique concerne la présence de l'opcode NOP, 0x00. La mémoire étant une ressource rare, la probabilité de présence volontaire d'instructions inutiles comme un NOP est très faible. Nous avons défini un seuil de 3% au-delà duquel la probabilité d'avoir une telle instruction dans une zone de code est faible. À partir de ces trois heuristiques notre algorithme peut détecter dans la masse de données disponibles la probabilité d'avoir une zone significative représentant une classe ou une instance d'objet.

5.3 Évaluation des contre-mesures

Le Class Loader dans la carte peut facilement détecter les modifications élémentaires dans le fichier CAP. Par exemple, lorsque dans le composant Reference Location, nous éliminons une entrée (entrée mise à zéro) sans recalculer l'offset de l'entrée suivante, certaines cartes rejettent ce code. La carte a-21a se bloque lorsqu'elle détecte un offset nul. Une modification plus élaborée du fichier CAP est alors nécessaire. Au moins trois des cartes évaluées disposent d'une forme de vérification de type incorporée au chargeur de classe. Ces cartes sont capables de détecter des suites d'instructions mal formées renvoyant par exemple une référence à la place d'un short. Après lecture de rapport de certification de Critères Communs il s'avère qu'effectivement ces cartes ne sont pas dans nos hypothèses. De telles cartes peuvent être considérées comme très sûres car dès que le byte code est détecté la carte devient muette et inutilisable, ce qui limite terriblement les capacités d'essai de l'attaquant. Pour les cartes ayant passé la phase de chargement, nous pouvons évaluer les différentes contre-mesures réalisées par l'interpréteur. Certaines cartes incorporent des mécanismes de gestion

mémoire tels que l'adresse de lecture ne correspond pas à l'adresse physique. Ainsi, lorsque nous tentons d'écrire à l'adresse lue, parfois nous obtenons des écritures dans d'autres zones. Une autre contre-mesure consiste à interdire l'écriture dans certaines zones de la mémoire eeprom. Ainsi la carte c-22a ne nous autorise pas à lire plus de sept octets consécutifs à l'aide d'un getstatic. La carte b-22b ne nous autorise que la lecture des zones de stockage des structures de classe. Le tableau ci-après (Table 1.) résume les capacités de notre attaque sur les différentes cartes.

Tab. 2. Comparaison de la capacité de l'attaque EMAN sur l'eeprom

Référence	Lecture	Écriture	Zone accessible
a-21a	x	x	8000-FFFF
a-21b	x	x	8000-FFFF
a-22a	x		8000-FFFF
a-22b	x		8000-FFFF
b-21a	x	x	8000-BFFF
b-22a	x	x	8000-BFFF
b-22b	x	x	8000-FFFF
c-22a	x		Sept octets
c-22b			

Notre attaque dans sa version actuelle nous permet de lire et d'écrire sur les zones eeprom de plusieurs cartes. Certaines nous autorisent que la lecture, et d'autres limitent la lecture à une zone de faible profondeur. Il nous reste à explorer la possibilité d'utiliser la pile C pour détourner le processeur vers un code natif.

6 Conclusion

Nous avons présenté dans ce papier une synthèse de l'attaque EMAN, de ses possibilités et des contre-mesures rencontrées. Cette attaque est basée sur une idée originelle de Hyppönen décrite dans sa thèse. Nous avons étendu et implémenté le concept avec un cheval de Troie pour une applet Java Card. Nous avons deux hypothèses de base i) le chargement post-issuance est autorisé et nous avons les clés de chargement ii) il n'y a pas de vérifieur de type embarqué. Nous avons démontré que cette attaque théorique était possible. Afin de pouvoir mener cette attaque, nous avons développé une méthodologie pour le chargement de code, une librairie d'accès (OPAL [1]) en Java implémentant Global Platform, une librairie pour la manipulation

[1] http://gforge.inria.fr/projects/opal/

de fichiers CAP et un outil d'analyse de plan mémoire. Notre attaque a été testée sur différentes cartes, certaines résistent bien grâce aux contre-mesures efficaces mises en place. Ces contre-mesures placent essentiellement ces cartes en dehors de nos hypothèses de travail (une forme réduite de vérification de type au chargement). Ce sont les cartes les plus récentes qui sont les mieux protégées contre cette attaque. Il nous reste à développer la partie sur la modification de la pile d'appel C, en détournant le compteur de programme vers une zone exécutable qui sera chargée comme un tableau dans une applet Java. Ce tableau contiendra le code écrit en natif pour le processeur cible (8051) de dump de la mémoire. Cette attaque ne sera réalisable que sur les composants pour lesquels la pile d'appel est située dans la mémoire et non sur un composant matériel spécifique. La question essentielle porte sur l'efficacité réelle et la portée de cette attaque. Notre cadre est bien celui des cartes de développement et non des cartes en production pour lesquelles le chargement post-issuance est souvent interdit. Si le chargement de code est autorisé alors il est fait généralement sous le contrôle du fournisseur d'application qui n'a aucun intérêt à placer un code malveillant dans une carte. Le second point qui tempère la portée de cette attaque est l'arrivée prochaine des cartes de type Java Card 3.0 connected edition, pour lesquelles le format de chargement est celui du fichier class et la présence d'un vérifieur de type est obligatoire. Dès que ces cartes seront en production, cette attaque deviendra inefficace. Ce travail a représenté néanmoins un défi technique pour les étudiants de notre master Cryptis [2] qui ont su implémenter un concept et parfois contourner les contre-mesures présentes dans les cartes.

Références

1. Joint interpretation library application of attack potential to smartcards (2006)
2. Virtual machine specification, java card platform, version 3.0, classic edition (2008)
3. Anderson, R., Kuhn, M. : Tamper resistance : a cautionary note. In : WOEC'96 : Proceedings of the 2nd conference on Proceedings of the Second USENIX Workshop on Electronic Commerce, p. 1. USENIX Association, Berkeley, CA, USA (1996)
4. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C. : The sorcerer's apprentice guide to fault attacks. Proceedings of the IEEE 94(2), 370–382 (2006). DOI 10.1109/jproc.2005.862424
5. Girard, P., Lanet, J.L. : New security issues raised by open cards. Information Security Technical Report 4(1), 4–5 (1999)
6. Hyppönen, K. : Use of cryptographic codes for byte code verification in smart card environment. Master's thesis, University of Kuopio (2003)
7. Mostowski,W., Poll, E. : Malicious code on java card smartcards : Attacks and countermeasures. In : Proc. Smart Card Research and Advanced Application Conference (CARDIS 2008), pp. 1–16 (2008)

[2] http://www.cryptis.fr

8. Vertanen, O. : Java Type Confusion and Fault Attacks, Lecture Notes in Computer Science, vol. 4326/2006, pp. 237–251. Springer Berlin, Heildeberg (2006)
9. Witteman, M. : Smartcard security. Information Security Bulletin 8, 291–298 (2003)