Escalade de privilège dans une carte à puce Java Card

Guillaume Bouffard[1] et Jean-Louis Lanet[2]

[1] Équipe Smart Secure Devices (SSD) – XLIM UMR 7252/Université de Limoges
123 avenue Albert Thomas, 87060 Limoges, France
guillaume.bouffard@unilim.fr
[2] Université de Limoges
123 avenue Albert Thomas, 87060 Limoges, France
jean-louis.lanet@unilim.fr

Résumé La carte à puce est aujourd'hui considérée comme étant un système sécurisé. Toutefois, il est possible, via certains types d'attaques, d'obtenir de l'information. Dans cet article, nous présentons comment ces informations ont permis de découvrir l'implémentation utilisée par une carte à puce pour appeler du code natif depuis le monde Java Card. Une fois ce mécanisme appréhendé, nous l'avons exploité pour exécuter notre propre code natif. Ce code, exécuté avec les droits du système d'exploitation, nous a permis de lire la mémoire ROM et ainsi obtenir tous les secrets de la carte attaquée. Une évaluation de cette attaque sur des cartes de modèles différents est présentée en conclusion.

Mots-clés: Java Card, Rétroconception, Système d'exploitation, Bac à sable, Attaque logique

1 Introduction

Quotidiennement, nous utilisons tous des cartes à puce pour payer, voyager, téléphoner... La carte à puce a su conquérir de nombreux marchés. Une des raisons de son succès vient du fait qu'elle est considérée comme un ordinateur sécurisé à bas coût. Ce système doit en effet garder secret tout son contenu (codes et données) et réaliser des traitements sans rien dévoiler. De par la sensibilité des applications présentes dans les cartes à puce, elles sont une cible privilégiée des attaquants. Leur objectif est alors d'obtenir le plus d'informations possible sur le fonctionnement de la carte.

Dans une carte à puce, le bien le plus important est le contenu de la zone ROM. Cette partie contient le code du système d'exploitation, les APIs et les applets Java Card sensibles. Le système d'exploitation contrôle chaque accès à cette zone afin d'empêcher un utilisateur malveillant

d'obtenir des biens de la carte. Actuellement, les meilleures attaques permettent d'obtenir seulement le contenu des fragments RAM et EEPROM.

Dans cet article, nous allons décrire comment sortir du bac à sable Java Card afin d'accéder à la zone ROM. Afin de mieux comprendre cette attaque, nous allons tout d'abord présenter la technologie Java Card dans la section 2. Ensuite, dans la section 3, nous allons décrire comment, au travers d'une attaque, il est possible d'obtenir un instantané de la mémoire d'une carte à puce. Après une analyse d'un plan mémoire dans la section 4, nous allons présenter une nouvelle attaque, section 5, permettant d'exécuter du code natif malicieux. Avant de conclure cet article, nous évaluerons notre attaque sur différentes cartes provenant de fournisseurs distincts dans la section 6.

2 La Technologie Java Card

Java Card est le portage de la technologie Java pour des systèmes embarquant très peu de ressources comme les cartes à puce. Cette technologie est basée sur les éléments de sécurité de Java et garde la simplicité de développement offert par ce langage. De plus, l'interopérabilité des applications Java Card est assurée par Oracle au travers de la spécification Java Card 3 édition classique [11].

2.1 Java

Java tire sa force de son architecture qui offre une interopérabilité fonctionnelle de la plate-forme d'exécution. La spécification de la technologie Java [10] assure que, peu importe l'implémentation de la Machine Virtuelle Java (MVJ), l'application exécutée aura toujours le même comportement. La figure 1 présente son modèle de fonctionnement.

Comme nous pouvons le voir dans cette figure, la MVJ est divisée en trois parties :

- l'environnement d'exécution Java,
- les bibliothèques Java,
- la *Java Native Interface* (JNI).

- L'environnement d'exécution Java : cet élément contient l'interpréteur de *bytecode* Java et les éléments de sécurité. Dynamiquement,

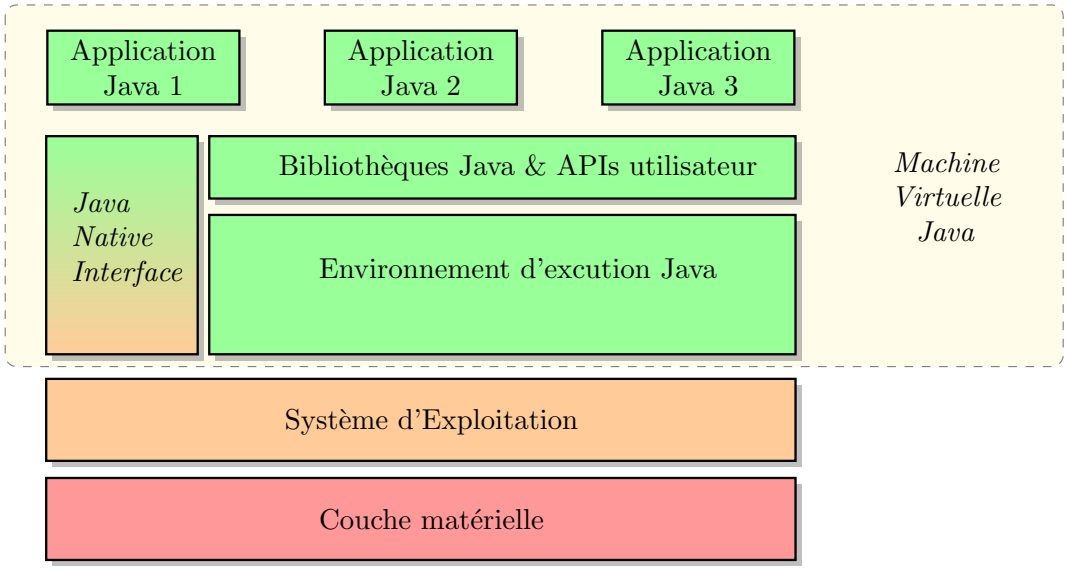


Figure 1. Modèle de fonctionnement Java.

chaque fichier de classe Java chargé est sémantiquement et structurellement vérifié par le chargeur de classes. Un fichier Java incorrect ne sera donc pas chargé.

Durant l'exécution, la plate-forme Java s'assure, via le contrôleur d'accès, que la requête sur chaque ressource est autorisée. Si ce n'est pas le cas, une exception est levée. Enfin, l'environnement d'exécution embarque un dernier élément de sécurité qui est le gestionnaire de sécurité. Ce composant joue le rôle de gardien entre le cœur de l'API Java et le système d'exploitation. Il a la responsabilité d'autoriser ou non l'accès aux ressources de l'hôte.

- **Les bibliothèques Java :** cette partie est composée de toutes les classes Java nécessaires au bon fonctionnement des applications exécutées. En plus de l'API standard spécifiée par Oracle et des diverses bibliothèques propriétaires fournies avec la MVJ, l'utilisateur peut en inclure d'autres suivant ses besoins. Il est laissé au choix de l'environnement d'exécution de vérifier la conformité de ces classes Java aux règles de sécurité Java.

- ***Java Native Interface* (JNI) :** durant l'exécution d'une application Java, il peut être nécessaire d'invoquer des fonctions (généralement natives) fournies par le système. Intégrée nativement dans l'environnement de développement Java, cette interface permet à l'application

d'appeler et d'être appelée par des éléments extérieurs.

Dans le monde Java, une méthode est invoquée via un appel à une instruction de type `invoke` suivie de la référence de la méthode demandée. Une méthode Java est composée d'un en-tête, correspondant à la structure `method_info`, et d'un tableau d'octets correspondant à l'ensemble de ses instructions. La référence d'une méthode correspond à son en-tête. L'entête est composé de son nom, de la description complète de la méthode (nom du paquetage, nom de la classe, ses champs...) et du type de la méthode, défini par le champ `access_flags`, de taille 2 octets. Cette structure est expliquée dans le listing 1.

```
method_info {
  // Masque indiquant le type d'accès et les propriétés de la
  // méthode.
  u2              access_flags;
  // Index vers le nom de la méthode.
  u2              name_index;
  // Index vers la description de la méthode.
  u2              descriptor_index;
  // Nombre d'attributs de la méthode.
  u2              attributes_count;
  // Liste des attributs de la méthode.
  attribute_info attributes[attributes_count];
}
```

Listing 1. En-tête d'une méthode Java.

Les valeurs possibles du champ `access_flags` sont décrites dans la table 1. Si `access_flag` contient la valeur `ACC_NATIVE`, le contenu de cette méthode sera donc une méthode native.

Si une méthode native est invoquée, une résolution par nom est faite par la MVJ. La spécification JNI [9] décrit une résolution au travers d'une table d'indirection.

En rapport avec le nom de la fonction demandée, la MVJ va parcourir la table des fonctions JNI contenant la liste des méthodes natives que l'application en cours d'exécution peut invoquer. Dès que la méthode appelée est trouvée, la MVJ va sortir du monde Java pour exécuter la partie de programme natif. Une fois ce fragment de code exécuté, la MVJ rend la main au programme Java qui poursuivra son exécution [3].

3. Il est à la charge de la partie native de faire en sorte que son état de retour soit utilisable par l'application Java.

Nom du champ	Valeur	Interprétation
ACC_PUBLIC	0x0001	Méthode publique : accessible à l'extérieur du paquetage.
ACC_PRIVATE	0x0002	Méthode privée : accessible seulement à l'intérieur de la classe.
ACC_PROTECTED	0x0004	Méthode protégée : accessible au travers des sous-classes.
ACC_STATIC	0x0008	Méthode statique.
ACC_FINAL	0x0010	Méthode `final` ; ne doit pas être surchargée.
ACC_SYNCHRONIZED	0x0020	Méthode synchronisée.
ACC_BRIDGE	0x0040	Méthode *bridge*, générée uniquement par le compilateur.
ACC_VARARGS	0x0080	Méthode avec un nombre d'argument variable.
ACC_NATIVE	0x0100	Méthode native : implémentée dans un langage autre que Java.
ACC_ABSTRACT	0x0400	Méthode abstraite : aucune implémentation n'est fourni.
ACC_STRICT	0x0800	Méthode déclarée comme `strictfp`.
ACC_SYNTHETIC	0x1000	Méthode déclarée comme `synthetic` ; la méthode n'est pas présente dans le code source, mais a été créée par le compilateur.

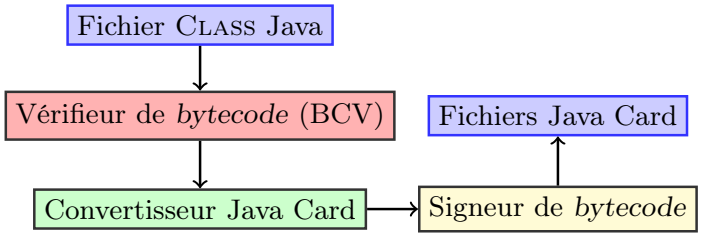
Table 1. Description du champ `access_flag` [10].

2.2 Java Card

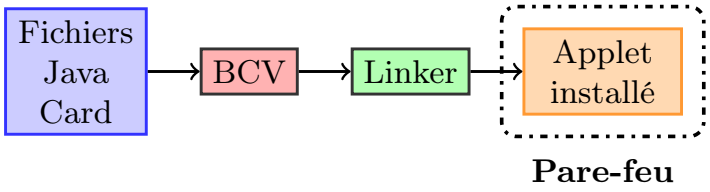
La plate-forme Java Card est un environnent multi-applicatif où les données critiques d'un applet doivent être protégées contre les accès malveillants provenant d'autres applications. Afin de protéger les applets entre eux, la technologie Java classique utilise la vérification de type, le chargeur de classes et le gestionnaire de sécurité pour créer un espace privé pour chaque applet. Dû aux fortes contraintes des cartes à puce, embarquer ces éléments de sécurité est très difficile. La solution adoptée est d'extraire la vérification hors de la carte. Dans la carte, le chargeur de classes et le gestionnaire de sécurité sont remplacés par le pare-feu Java Card.

Le modèle de sécurité Java Card repose donc sur deux parties : l'une déportée et l'autre embarquée dans la carte. Hors de la carte (figure 2(a)), les fichiers Class Java doivent être convertis en un format de fichier (le format Cap) optimisé pour les périphériques avec très peu de ressources comme la carte à puce. Durant cette phase de conversion, le programme doit respecter les règles du langage Java. Ceci est vérifié par le vérifieur de *bytecode*. Le fichier de sortie peut être signé pour garantir sa provenance et son intégrité.

Pour des raisons de sécurité, la possibilité de télécharger du code dans la carte est contrôlée par un protocole définit par GlobalPlatform [5]. Ce protocole s'assure que le propriétaire du programme a les droits nécessaires.



(a) Modèle de sécurité hors de la carte.



(b) Modèle de sécurité embarqué dans la carte.

FIGURE 2. Modèle de sécurité Java Card.

Une fois l'applet installé, le pare-feu Java Card ségrègue chaque applet installé dans la carte (figure 2(b)).

3 Extraction d'un plan mémoire Java Card

L'un des principaux biens protégés par une carte à puce est le contenu de sa mémoire, contenant les clefs cryptographiques, le code des applications... Découvrir une vulnérabilité dans une application permet d'obtenir des informations sur le code et/ou les données contenues dans la plate-forme attaquée.

Pour arriver à ses fins, il existe trois approches : les attaques physiques, les attaques logiques et les attaques combinées qui mêlent les attaques physiques aux attaques logiques. Dans cet article, nous nous concentrerons sur les attaques logiques et combinées.

3.1 Les Attaques Logiques

- Exploitation de la spécification : Hubbers *et al.* [7] ont présenté différents moyens de réaliser une confusion de type. Une manipulation du fichier CAP après la phase de conversion permet d'outrepasser le vérifieur de *bytecode* et l'injection d'une faute permet d'abuser d'un potentiel vérifieur embarqué.

Dans le cas où on ne peut outrepasser le vérifieur de *bytecode*, les auteurs ont proposé un moyen d'abuser du mécanisme de transaction tout en gardant une application correcte pour le vérifieur de *bytecode*. Le mécanisme de transaction permet de réaliser des opérations atomiques dans la carte. D'après la spécification [11], l'annulation d'une transaction désalloue chaque objet créé durant la transaction et chaque référence réinitialisée doit avoir la valeur `null`. Cependant, les auteurs ont découvert des cas où la carte conserve la référence des objets créés lorsqu'une transaction est annulée. Cette référence non nulle est associée à un élément sur le tas qui n'existe plus. Lors des prochaines allocations sur le tas, un objet de type différent est alloué au même endroit sur le tas. Il existe alors deux pointeurs, de type différent, associés à cet élément.

Les auteurs ont également mis en avant qu'il est possible d'abuser du mécanisme de partage. Cette attaque n'est actuellement plus possible sur les cartes récentes.

- Absence de vérification dans la carte : Lors de SSTIC 2009, Iguchi-Cartigny *et al.* [8] ont décrit un moyen de contourner le pare-feu Java Card. D'après la spécification [11], le pare-feu ne contrôle pas l'utilisation des opérations sur les valeurs statiques. Pour interagir avec les éléments statiques, la spécification Java Card [11] définit trois instructions : `getstatic`, `putstatic` et `invokestatic`. Dans un fichier CAP malveillant, c'est-à-dire modifié avant d'être chargé sur une carte, le paramètre de l'instruction `invokestatic` est changé afin de rediriger le flux d'exécution dans un autre applet. De plus, grâce à l'instruction `getstatic`, les auteurs ont réussi à lire les mémoires RAM et EEPROM d'un certain nombre de cartes à puce. L'instruction `putstatic`, quant à elle, permet d'écrire en mémoire.

Une autre partie attaquée dans la carte est la valeur de l'adresse de retour d'une fonction. D'après la spécification de la MVJ [10], l'adresse de l'appelant (ou adresse de retour) d'une méthode est stockée dans l'en-tête de la *frame* courante. La spécification Java Card [11] ne définit rien à ce sujet. Toute cette partie est donc laissée au choix du développeur de la Machine Virtuelle Java Card (MVJC). Bouffard *et al.* [2] ont découvert que, comme la MVJ, cette adresse est stockée dans l'en-tête de la *frame* courante. Dans la *frame* d'une méthode Java Card, l'en-tête se situe entre la pile des variables locales et la pile des éléments. Au travers d'un *buffer overflow*, les auteurs ont montré comment modifier l'adresse de retour afin

d'exécuter le contenu d'un tableau. Récemment, Faugeron [4] a expliqué une attaque similaire grâce à un *buffer underflow*.

Hamadouche *et al.* [6] ont introduit les attaques sur le *linker* Java Card. Pour cela, ils ont proposé un moyen efficace d'obtenir les références internes de chaque élément de l'API. Pour y arriver, les auteurs chargent dans la carte un applet contenant des éléments considérés comme des symboles mais n'étant pas précédés par une instruction nécessitant des paramètres à résoudre. À cause de ses ressources limitées, le *linker* embarqué ne vérifie pas le type d'instruction qui précède le symbole à résoudre et, si la valeur à résoudre est valide, il le résout. Grâce à ça, les auteurs sont capables de récupérer les références internes de chaque fonction proposée par l'API embarquée dans la carte. Avec ces adresses, l'attaquant peut développer des *shellcodes* complexes exécutables sur le modèle de carte attaquée. Razafindralambo *et al.* [13] sont allés plus loin dans cette attaque en faisant muter un code structurellement correct en un applet malveillant en abusant du *linker* Java Card embarqué.

Bouffard *et al.* [3] ont montré qu'il était possible d'abuser du convertisseur Java Card pour lier un applet avec une librairie malveillante. Les auteurs se sont focalisés sur la phase de traduction des fichiers Class en fichier Cap. Durant cette étape, le convertisseur Java Card va chercher, pour chaque élément nommé du fichier Class, le lien permettant de traduire le nom (formé à partir d'une chaîne de caractères encodée en UTF-8) vers le symbole utilisé par la carte. Le convertisseur Java Card ne recherche que la première occurrence donnant le lien entre un nom et son symbole. Les auteurs ont montré qu'en incluant un mauvais lien vers une librairie malveillante ayant un comportement similaire à l'original, en apparence. Le convertisseur Java Card utilisera ce lien si c'est le premier trouvé. Cette attaque se rapproche d'une attaque de type *Man in the middle*.

3.2 Les Attaques Combinées

De nos jours, grâce à l'augmentation des ressources embarquées, il est devenu très difficile d'installer un applet mal formé dans une Java Card. En effet, les cartes récentes embarquent un vérifieur de *bytecode* qui vérifie statiquement l'application installée. Pour outrepasser cela et exécuter des programmes mal formés, les scientifiques expliquent [1,2,12] qu'une faute injectée peut dynamiquement modifier une application correctement installée et ainsi réaliser les attaques logiques précédemment citées.

4 Analyse d'un plan mémoire Java Card

Grâce aux attaques présentées dans la section 3, nous avons réussi à obtenir un instantané d'une partie du plan mémoire d'une carte à puce Java Card. Ce plan mémoire est composé de la RAM et de l'EEPROM. Le modèle de virtualisation empêche la MVJC d'accéder à la ROM par un chemin non autorisé. Il n'est pas possible que la zone ROM (contenant le système d'exploitation, l'API et les applets Java Card critiques) soit lue sans l'autorisation du système d'exploitation. L'attaque permettant d'obtenir cet instantané du plan mémoire a nécessité la connaissance des clefs de chargement et l'installation d'un applet mal formé. Depuis cet applet malveillant, la mémoire est lue par fragment de 255 octets [4]. Chaque octet lu a été stocké dans le tampon APDU. Ce tampon est ensuite envoyé au lecteur.

Actuellement, le reverse d'un plan mémoire d'une carte à puce est réalisé manuellement. C'est un travail long, difficile et fastidieux souvent source d'erreurs. Pour automatiser le reverse de la mémoire d'une Java Card, nous avons développé un outil, nommé *Java Card DisAssembler* (JCDA).

Dans cette section, nous étudions une carte de développement embarquant une Machine Virtuelle Java Card 2.1, Visa Open Platform (OP) 2.0 et un processeur 8 bits. La carte cible contient un CPU cadencé à 5MHz avec 32ko de ROM, 32ko d'EEPROM et 2ko de RAM. De plus, nous connaissons les clefs de chargement de cette carte. La MVJC est stockée en zone ROM et est conforme à la spécification [11]. Le plan mémoire de la carte analysée est décrit dans la figure 3.

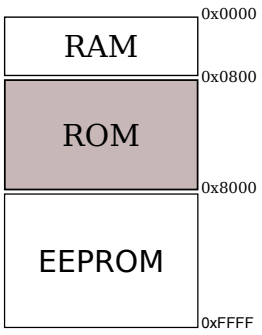


Figure 3. Organisation de la mémoire sur la carte analysée.

4. Cette fenêtre est de la même taille que celle du tampon APDU.

4.1 Stockage d'un applet Java Card dans une carte à puce

Le fichier CAP est basé sur la notion de composants interdépendants. Il est spécifié dans [11] et il est constitué de onze composants standards : `Header`, `Directory`, `Import`, `Applet`, `Class`, `Method`, `Static Field`, `Export`, `Constant Pool`, `Reference Location` et `Descriptor`. De plus, certaines MVJC peuvent supporter des composants personnalisés. Lors de la phase d'installation, chaque composant nécessaire est stocké dans la mémoire de la carte. Comme expliqué par [8], sur la carte étudiée, une application installée peut être structurée comme dans la figure 4.

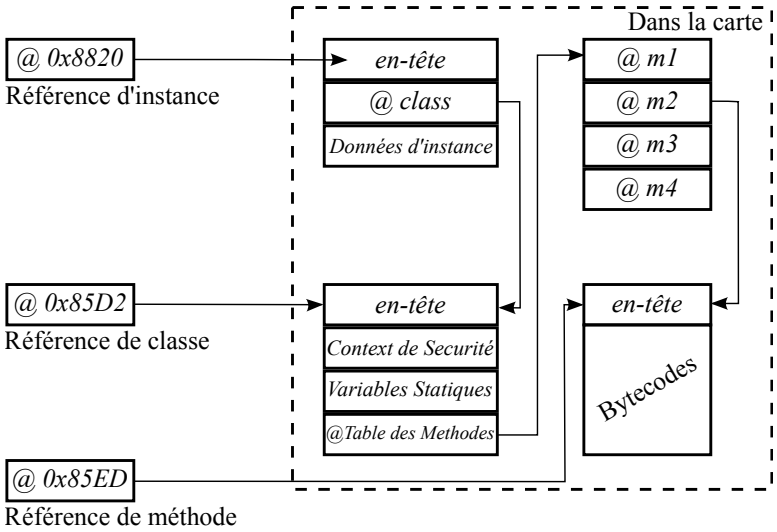


FIGURE 4. Représentation d'un applet installé dans la mémoire d'une Java Card.

4.2 Découverte d'un comportement non spécifié

Durant l'analyse de chaque applet installé dans la mémoire de la carte, nous avons découvert une méthode qui appelle un élément non standard à l'adresse `0xDBE6`. Cette méthode est présentée dans le listing 2.

```
0xDBE6: 01 // flags: 0 max_stack : 1
0xDBE8: 00 // nargs: 0 max_locals: 0
0xDBE9: 8D DB C7 invokestatic DB C7
0xDBEC: 67 08    ifnonnull    08
0xDBEE: 11 6F 00 sspush       6F 00
0xDBF0: 8D 6F 05 invokestatic 6F 05 // ISOException.throwIt(0x6F00);
0xDBF3: 7A       return
```

Listing 2. Méthode appelant une fonction non standard.

À l'adresse `0xDBE9`, l'instruction `invokestatic` référence une méthode dans la zone EEPROM située à l'adresse `0xDBC7`. À cette adresse, plusieurs méthodes non standards ont été trouvées, comme présenté dans la table 2.

Table 2. Table des méthodes non standards.

0xDBC4	21 01 32	0xE684	21 00 40
0xDBC7	24 00 33	0xE687	21 00 41
0xDBCA	24 00 34	0xE68A	21 00 42
0xDBEA	22 01 35	0xE024	22 00 43
0xDBF9	22 00 36	0xE69C	21 02 44
0xDF7D	21 02 37	0xE69F	21 03 45
0xE66C	24 01 38	0xE6A2	22 01 46
0xE66F	24 00 39	0xF251	24 01 47
0xE672	21 00 3A	0x96BC	23 00 48
0xE675	24 00 3B	0xF32D	22 01 49
0xE678	24 00 3C	0xF330	22 02 4A
0xE67B	22 00 3D	0xF7B5	24 02 4B
0xE67E	24 04 3E		
0xE681	21 00 3F		

La spécification de MVJC définit une méthode comme une structure ayant un en-tête de type `method_header_info`, décrit dans le listing 3, et son *bytecode*.

```
method_header_info {
  u1 bitfield {
    bit[4] flags // modificateurs de la méthode
    bit[4] max_stack // nombre maximum de mots nécessaires,
                     // sur la piles des opérandes, à l'exécution
                     // de la méthode
  }

  u1 bitfield {
    bit [4] nargs // nombre de paramètres de la méthode
    bit [4] max_locals // nombre de variables locales
                       // déclarées par la méthode
  }
}
```

Listing 3. Structure de l'en-tête d'une méthode Java Card.

Pour le champ `flag`, trois valeurs sont possibles :
- `0x0` : dans le cas d'une méthode normale ;
- `0x8` (`ACC_EXTENDED`) : la méthode courante est une méthode étendue ;
- `0x4` (`ACC_ABSTRACT`) : la méthode courante est une méthode abstraite ;

– Toutes les autres valeurs sont réservées pour de futures implémentations.

Chaque méthode listée dans la Table 2 est constituée d'une valeur de `flag` non standard (`0x2`). De plus, le *bytecode* associé ne peut pas correspondre au code d'une fonction. Il est trop court pour cela.

De plus, grâce au JCDA, nous avons découvert une table contenant une suite d'adresses, donnée dans la table 3. Nous supposons que toutes les valeurs sont des adresses qui référencent un élément dans la zone ROM (la valeur est inférieure `0x8000`) excepté une adresse colorée en gris (`0xFF5C`).

Table 3. Liste d'adresses dans la zone EEPROM.

	0x0	0x2	0x4	0x6	0x8	0xA	0xC	0xE
0x008060								58 00
0x008070	7E 84	6A DC	6A ED	6A FE	18 00	7F 08	7F 29	7F 02
0x008080	7F 24	7E FC	5E 79	47 AD	67 32	6B 85	49 DD	68 BD
0x008090	5F 9F	5D C9	63 1D	46 38	5E A5	7E 01	0F EB	69 15
0x0080A0	6C 22	68 A7	5F EF	6B 0F	6B 20	6B 31	6B 42	6B 53
0x0080B0	7E F0	38 BE	62 D9	57 67	6B 64	4E A3	55 DA	7F 31
0x0080C0	7F 46	52 08	37 8F	FF 5C	65 15	5C E5	7E F6	67 C7
0x0080D0	7F 1A	63 A0	67 32	49 DD	7E A2	61 E4	64 1F	67 AF
000080E0	37 FB	69 2A	67 32	17 FB	7E 7A	48 7C	16 86	7D E9
0x0080F0	7F 35	7E EA	7F 1F	5A EA	2A AC	7D 9C	7E E4	7D 8F
0x008100	61 E4	67 F7	27 97	64 D9	60 00	00 9C	00 9E	00 00

Pour confirmer notre hypothèse, nous avons vérifié ce qui est contenu à l'adresse `0xFF5C` afin de comprendre la signification de cette table. Comme nous pouvons l'observer dans le listing 7 (annexe A), le fragment de code listé est une méthode composée d'instructions compilées en assembleur 8051. Ce langage correspond au langage bas niveau de la plate-forme analysée.

4.3 JNI et Java Card : la table d'indirection

La spécification Java Card [11] ne définit pas d'interface JNI. Il est toutefois probable que cela existe. En effet, la partie cryptographique est exécutée via des fonctions nécessitant une partie native et sont généralement exécutées sur un co-processeur cryptographique.

Pour exécuter des fonctions natives, la MVJC doit utiliser une table d'indirection. Ce type de table est utilisé pour passer du monde Java Card au monde natif. Supposons avoir une instruction de type `invokestatic` qui appelle une méthode native, la méthode appelé aura une valeur de `flag` égale à `0x2`. Cette méthode non standard référence un élément dans la table d'indirection permettant d'obtenir l'adresse de la méthode native associée. Nous avons, dans cette implémentation de la MVJC, un fonctionnement proche des JNI. Cette procédure est décrite dans la figure 5.

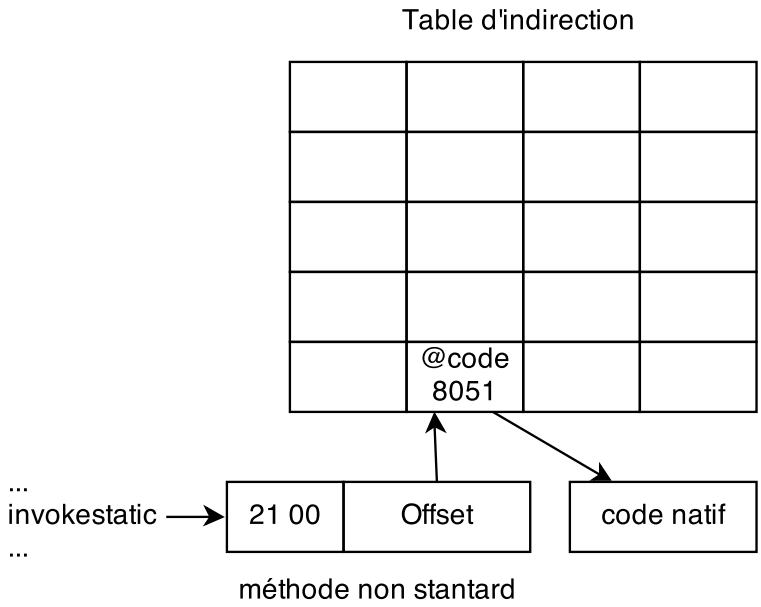


Figure 5. Utilisation d'une table d'indirection.

5 EMAN3 : tous les chemins mènent à la ROM

À présent que nous savons comment fonctionne le mécanisme de JNI sur Java Card, voyons comment nous pouvons l'exploiter pour exécuter notre code natif.

Comme décrit dans la figure 5, l'implémentation propriétaire de la MVJC (une méthode avec une valeur de `flag` égale à `0x2`) contient un offset de la table d'indirection.

Chaque élément dans la table d'indirection (table 3) référence une fonction native. Pour corrompre le mécanisme de redirection, nous allons

rajouter un nouvel élément dans la table d'indirection contenant l'adresse de notre *shellcode* natif. En appelant ce nouvel élément au travers d'une méthode contenant un `flag` à la valeur `0x2`, il nous sera alors possible d'exécuter du code natif. L'objectif ici est d'accéder aux données stockées en zone ROM, accessibles uniquement par le système d'exploitation.

Pour accéder à la zone ROM, nous allons diviser la modification en trois étapes. Tout d'abord, nous allons créer une fonction, nommée `callNative()`, contenant une valeur de `flag` à `0x2`. Cette méthode contient un seul octet correspondant à l'offset de l'adresse de notre *shellcode* dans la table d'indirection. Cette méthode est présentée dans le listing 4.

```
callNative() {
0x21 // flags: 2 max_stack : 1
0x00 // nargs: 0 max_locals: 0
YY   // Offset contenant l'adresse du shellcode dans la table
     // d'indirection
}
```

Listing 4. Notre méthode Java frauduleuse.

Ensuite, nous créons un tableau, nommé `native_shellcode`, qui contient notre *shellcode* développé en assembleur 8051.

Pour mettre à jour la table d'indirection, il nous faut l'adresse du tableau `native_shellcode`. Pour la découvrir, nous allons utiliser une confusion de type [8] comme celle décrite dans le listing 5.

```
short giveObjectAddress(Object object) {
0x01 // flags: 0 max_stack : 1
0x10 // nargs: 1 max_locals: 0
0x19 aload_1
0x78 sreturn
}
```

Listing 5. Confusion de type simple.

Dans cet exemple, la référence de l'objet `object` donnée en paramètre est poussée en haut de la pile des éléments via l'instruction `aload_1`. L'instruction `sreturn` retourne, quant à elle, le dernier entier court poussé sur la pile. Sur la carte analysée, les références sont encodées sur un entier court (de type `short`). Pour un `object` donné, la fonction `giveObjectAddress()` convertit la référence de cet objet en une valeur numérique correspondant à so adresse. En utilisant cette fonction, nous pouvons obtenir l'adresse de notre tableau `native_shellcode`.

Enfin, pour exécuter notre *shellcode*, nous devons mettre à jour la table d'indirection avec l'adresse du contenu du tableau `native_shellcode`. Avec la connaissance de son adresse, via la fonction `giveObjectAddress()`, nous rajoutons, à la fin de la table (adresse `0x810E` de la table 3), l'adresse de notre *shellcode*. L'offset de la fonction `callNative()` doit, en effet, référencer l'élément rajouté dans la table d'indirection. Dans notre cas, cet offset sera égal à `0x50`.

Pour lire le contenu de la zone ROM, nous initialisons notre tableau `native_shellcode` par le *shellcode* listé dans le listing 8 (Annexe B) et nous exécutons la fonction `callNative()`. En indiquant l'adresse de début de la zone à lire, le *shellcode* natif lit chaque octet présent et l'écrit dans le tampon APDU. Grâce à ça, nous avons réalisé une escalade de privilège en exécutant du code natif hors du bac à sable Java Card ce qui nous offre un accès complet à tous les secrets de la carte analysée.

6 Évaluation de notre attaque : un exploit générique ?

Nous avons évalué notre attaque sur différents modèles de cartes provenant de différents encarteurs. Les cartes évaluées sont toutes disponibles publiquement via internet. Nous avons évalué sept 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. Nous supposons que chacune des cartes évaluées supporte l'exécution de méthodes natives au travers de la MVJC. La liste des cartes est présentée dans la table 4.

Référence	Java Card	GlobalPlatform	Caractéristiques
`a-21a`	2.1.1	2.0.1	
`a-22b`	2.2	2.1	72kB EEPROM
`b-22a`	2.2.1	2.1.1	36kB EEPROM, RSA
`b-22b`	2.2.2	2.1.1	72kB EEPROM, RSA
`b-21c`	2.1.1	2.1.2	16kB EEPROM, RSA
`c-21a`	2.1	2.0.1	32KB EEPROM, RSA
`c-22b`	2.2.1	2.1.1	16kB EEPROM

Table 4. Définition des cartes utilisées dans cette étude.

Dans chaque carte, nous avons chargé un fichier CAP contenant la méthode listée dans le listing 6. Cette méthode contient un `flag` égal `0x2`. Le code de cette méthode contient un octet. Dans l'hypothèse où se

code est interprété comme un offset vers la table d'indirection, la première méthode native sera appelée.

```
callNative() {
  21 // flags: 2 max_stack : 1
  01 // nargs: 0 max_locals: 1
  // Offset correspondant à un élément dans la indirection table
  00 NOP // Cette instruction ne fait rien (Do Nothing)
}
```

Listing 6. Méthode appelant une fonction native utilisée pour l'évaluation de l'attaque EMAN3.

La méthode contenue dans le listing 6 est exécutée sur toutes les cartes évaluées. Le retour de chaque carte est présenté dans la table 5.

Référence	Valeur de retour
a-21a	Code de retour : 0x6F00
a-22b	Erreur PCSC : SCARD_E_NOT_TRANSACTED
b-22a	Code de retour : 0x9000
b-22b	Code de retour : 0x9000
b-21c	Code de retour : 0x6F00
c-21a	Code de retour : 0x6F00
c-22b	Erreur PCSC : SCARD_E_NOT_TRANSACTED

Table 5. Valeur de retour de chacune des cartes testées pour l'appel de la méthode native listée dans le listing 6.

Dans cette table, trois valeurs différentes sont retournées :
- La valeur 0x9000 : l'exécution s'est déroulée sans erreur.
- La valeur 0x6F00 : une exception Java non définie à été lancée. Cela est dû à une erreur Java non attrapée.
- Une erreur PCSC. La carte à un comportement anormal durant la connexion avec l'hôte et la communication est stoppée lors de l'exécution d'une commande (SCARD_E_NOT_TRANSACTED).

Dans le cas où une valeur d'erreur est retournée, nous ne pouvons rien en conclure.

Dans la table 5, deux cartes (b-22a et b-22b) retournent le code de retour 0x9000. Arbitrairement, nous avons choisi d'étudier la carte b-22a.

Pour compléter notre analyse, nous allons rechercher comment appeler la fonction `arrayCopy()` [5]. Cette fonction est fournie par la classe `Util` appartenant au paquetage `javacard.framework` et est probablement développée en C. Dans notre application, nous appelons une méthode qui a la même signature que la fonction `arrayCopy()` où sa valeur de `flag` est mis à `0x02`. Grâce à un script, nous recherchons la valeur de l'offset qui appelle une méthode native correspondant au retour attendu. Après avoir fait varier cet offset de `0x00` à `0xFF`, aucun retour correct n'a été découvert.

En investiguant plus, nous avons découvert que sur la carte `b-22a`, l'offset de la méthode native à appeler est encodé dans l'en-tête de la méthode via l'octet modélisant les champ `nargs` et `max_locals` (*cf* listing 3). Sur la carte `b-22b`, nous avons réussi à appeler une méthode native permettant de modifier le cycle de vie de la carte. En appelant la méthode ayant l'offset `0x0F`, la carte est passée en mode production [6] sans autorisation particulière. On peut constater ce changement de mode par le modification de l'ATR de la carte. Nous avons ici une preuve que nous sommes capables d'exécuter du code natif. Toutefois, sur cette carte, nous n'avons pas réussi à trouver où est stockée la table d'indirection. Elle est probablement masquée par un `xor`.

7 Conclusion

Dans cet article, nous avons présenté un chemin permettant d'exécuter du code natif dans une carte à puce Java Card. Grâce à une attaque, nous avons obtenu un instantané de la mémoire d'une carte. Au travers d'une étape d'ingénierie inverse, nous avons compris comment du code natif est appelé depuis le monde Java Card. Exploiter ce mécanisme nous a permis d'outrepasser le bac à sable Java, de lire la zone ROM et ainsi d'accéder à tous les codes de la carte à puce.

En évaluant notre travail sur d'autres cartes, nous avons découvert que le mécanisme attaqué est implémenté sur des cartes de fournisseurs et de modèles différents d'une manière sensiblement similaire. En trouvant où est situé la table d'indirection, et en ayant un moyen de la modifier, on pourrait tout à fait y réaliser notre attaque.

5. Cette fonction retourne une valeur correspondant à l'offset de début dans le tampon de destination plus la longueur des données copiées.

6. Le mode administration est verrouillé. Seuls les applets installés peuvent être utilisés.

Remerciements

Les auteurs souhaitent remercier Julien Boutet pour sa contribution majeure durant ce travail.

Références

1. G. Barbu. *De la sécurité des plateformes Java Card face aux attaques matérielles*. PhD thesis, Telecom ParisTech, September 2012.
2. G. Bouffard, J. Iguchi-Cartigny, and J.-L. Lanet. Combined Software and Hardware Attacks on the Java Card Control Flow. In E. Prouff, editor, *Smart Card Research and Advanced Applications*, volume 7079 of *Lecture Notes in Computer Science*, pages 283–296. Springer Berlin/Heidelberg, 2011.
3. Guillaume Bouffard, Tom Khefif, Jean-Louis Lanet, Ismael Kane, and Sergio Casanova Salvia. Accessing secure information using export file fraudulence. In Bruno Crispo, Ravi S. Sandhu, Nora Cuppens-Boulahia, Mauro Conti, and Jean-Louis Lanet, editors, *CRiSIS*, pages 1–5. IEEE, 2013.
4. E. Faugeron. Manipulating the frame information with an underflow attack. In *CARDIS 2013*, November 27th-29th 2013.
5. GlobalPlatform. Card specification, January 2011.
6. S. Hamadouche, G. Bouffard, J.-L. Lanet, B. Dorsemaine, B. Nouhant, A. Magloire, and A. Reygnaud. Subverting Byte Code Linker service to characterize Java Card API. In *Seventh Conference on Network and Information Systems Security (SAR-SSI)*, pages 75–81, May 22rd to 25th 2012.
7. E. Hubbers and E. Poll. Transactions and non-atomic API calls in Java Card : specification ambiguity and strange implementation behaviours. Dept. of Computer Science NIII-R0438, Radboud University Nijmegen, 2004.
8. J. Iguchi-Cartigny and J.-L. Lanet. Évaluation de l'injection de code malicieux dans une java card. In *Symposium sur la Sécurité des Technologies de l'Information et de la Communication, SSTIC*, pages 3–19, June 2009.
9. S. Liang. *The Java TM Native Interface : Programmer's Guide and Specification*. Addison-Wesley Professional, 1999.
10. T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification*. Addison-Wesley, 2013.
11. Oracle. Java card 3 platform, virtual machine specification, classic edition. (Version 3.0.4), September 2011.
12. T. Razafindralambo, G. Bouffard, and J.-L. Lanet. A friendly framework for hidding fault enabled virus for Java based smartcard. In N. Cuppens-Boulahia, F. Cuppens, and J. Garcia-Alfaro, editors, *Data and Applications Security and Privacy XXVI*, volume 7371 of *Lecture Notes in Computer Science*, pages 122–128. Springer Berlin/Heidelberg, 2012.
13. T. Razafindralambo, G. Bouffard, B. N Thampi, and J.-L. Lanet. A Dynamic Syntax Interpretation for Java Based Smart Card to Mitigate Logical Attacks. In S. M. Thampi, A. Y. Zomaya, T. Strufe, J. M. Alcaraz Calero, and T. Thomas, editors, *SNDS*, volume 335 of *Communications in Computer and Information Science*, pages 185–194, Trivandrum, India, 2012. Springer.

A Code natif découvert dans la zone EEPROM

```
FF5C              lcall code_5A46
FF5F              jnc   code_FF9A
FF61              clr   C
FF62              mov   A, RAM_3F
FF64              subb  A, #0x80
FF66              jnc   code_FF76
FF68              mov   R7, RAM_40
FF6A              mov   R6, RAM_3F
FF6C              mov   R4, RAM_76
FF6E              mov   R5, RAM_77
FF70              mov   R3, RAM_44
FF72              lcall code_3F6B
FF75              ret

FF76 code_FF76:   ; CODE XREF: code:0000↑FF66j
FF76              setb  RAM_20.0
FF78              jnb   RAM_20.2, code_FF8B
FF7B              clr   A
FF7C              mov   R7, A
FF7D              lcall code_6C04
FF80              mov   RAM_45, R6
FF82              mov   RAM_46, R7
FF84              mov   R7, #0x45 ; 'E'
FF86              lcall code_3CE3
FF89              mov   RAM_20.0, C

FF8B code_FF8B:   ; CODE XREF: code:0000↑FF78j
FF8B              mov   R7, RAM_40
FF8D              mov   R6, RAM_3F
FF8F              mov   R4, RAM_76
FF91              mov   R5, RAM_77
FF93              mov   R3, RAM_44
FF95              mov   R2, RAM_43
FF97              lcall code_2961

FF9A code_FF9A:   ; CODE XREF: code:0000↑FF5Fj
FF9A              ret
```

Listing 7. Fragment de code natif découvert dans la zone EEPROM.

B Code Natif permettant de lire la zone ROM

```
mov   DPTR,  @tab     ; Adresse du tableau transient ou
                      ; seront stocke les données lues.
;; Sauvegarde des registres A, R0 et R1
movx  @DPTR, A        ; A est sauvegarde
mov   DPTR,  @tab+1   ;
mov   A,     R0
movx  @DPTR, A        ; R0 est sauvegarde
mov   DPTR,  @tab+2
mov   A,     R1
movx  @DPTR, A        ; R1 est sauvegarde
```

```
mov   R0,    1        ; R0 est utilise comme offset

LOOP1:  ;; boucle principale ou la ROM est lue
  mov   A,     R0
  mov   R1,    A              ; Copie de A dans le registre R1
  mov   DPTR,  @ROM           ; Copie de la premiere adresse de la
                              ; ROM a lire
  movc  A,     @A + DPTR      ; incrementation de 1ere adresse
                              ; en ROM a lire
  mov   DPTR,  @BUFFER_APDU   ; déplacement de l'adresse du tampon
                              ; APDU vers le registre DPTR

 LOOPINC:  ;; Le registre DPTR est utilise comme un
           ;; index dans le tampon APDU
    inc   DPTR                ; Incrementation du registre DPTR
    djnz  R1, LOOPINC         ; Décrémente R1 tant qu'il est > 0

  movx   @DPTR, A             ; la valeur lue en ROM est
                              ; sauvegarde dans le tampon APDU

  inc   R0                    ; incrémente le R0
  mov   A,     R0             ; Met à jour l'offset lu
                              ; en ROM pour la prochaine itération
  jz LOOP1                    ; Saute vers LOOP1 si A != 0

;; Restaure les registres CPU
mov   DPTR,  @tab+2
movx  A,     @DPTR
mov   R1,    A        ; R1 est restaure
mov   DPTR,  @tab+1
movx  A,     @DPTR
mov   R0,    A        ; R0 est restaure
mov   DPTR,  @tab
movx  A,     @DPTR    ; A est restaure
ret
```

Listing 8. Lecture de 255 octets de la zone ROM en 8051.