Compromission d'une application bancaire JavaCard par attaque logicielle

Julien Lancia

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

Résumé Les plateformes Java Card permettent de garantir la sécurité des applications embarquées sur cartes à puces au moyen de mécanismes de sécurité factorisés au niveau de la machine virtuelle. Cependant, le caractère optionnel du mécanisme de vérification de bytecode jusqu'à la version 2.2.2 de Java Card permet de charger des applications malicieuses et de réaliser des attaques logiques sur ces plateformes. Cet article présente une attaque logique sur une plateforme Java Card intégrant de nombreux mécanismes de sécurité allant au delà des recommandations sécuritaires imposées par les spécifications Java Card. Malgré ces protections, nous montrons qu'une vulnérabilité d'apparence anodine permet de compromettre intégralement l'application bancaire embarquée sur la plateforme Java Card. Nous détaillons également deux contre-mesures permettant de se prémunir contre cette attaque.

Introduction

Les types d'applications auxquelles sont destinées les cartes à puce font de la sécurité un enjeu majeur de ce domaine. Les données bancaires, les informations médicales et biométriques stockées dans les cartes à puces, ainsi que les données personnelles stockées dans les cartes SIM des smartphones sont autant de biens à protéger pour les concepteurs de puces et les développeurs d'applications embarquées.

Pour être efficace, la sécurité d'un produit de type carte à puce doit être conçue de manière globale, depuis la couche matérielle jusqu'aux couches logicielles. Concernant la couche matérielle, de nombreux types d'attaques existent, parmi lesquelles les attaques par injection de faute [2,19,6] et les attaques par canaux cachés [15,1,11]. Les puces modernes offrent un certain nombre de contre-mesures pour se protéger contre ce genre d'attaques [16,5,18,12]. Les attaques logicielles (ou attaques logiques) consistent pour leur part à exploiter un bogue ou un défaut de conception de l'application pour mettre à défaut la sécurité du produit [14,9,13]. Les systèmes Java Card [21,22,23,20] permettent en théorie de se prémunir contre les attaques logiques en factorisant la sécurité dans la

machine virtuelle qui exécute l'application. La machine virtuelle est par exemple responsable des vérifications de débordement de tableaux ("buffer overflow") et de la séparation des domaines de sécurité ("firewall"). Les machines virtuelles défensives embarquées dans les cartes à puces récentes implémentent de nombreuses contre-mesures logicielles, allant au delà des vérifications imposées par les spécifications Java Card afin d'empêcher toute attaque logique.

L'attaque présentée dans cet article se base sur un concept d'attaque logique existant [8], et le pousse plus avant afin de l'appliquer au cas concret d'un produit Java Card embarquant une applet bancaire [1]. Bien que la machine virtuelle de ce produit implémente de nombreuses contre-mesures logiques, nous montrons qu'une vulnérabilité résiduelle d'apparence anodine permet de compromettre entièrement l'applet bancaire.

Le produit qui constitue la cible de notre attaque est un produit de type "plateforme ouverte", c'est à dire que le chargement d'applets additionnelles est autorisé durant la phase d'utilisation de la carte. Dans le cas contraire, les commandes d'administration sont définitivement désactivées lorsque la carte est distribuée au porteur. Le mécanisme de chargement d'applets sur une plateforme Java Card est généralement implémenté suivant la spécification GlobalPlateform [7], et dans ce contexte le chargement est strictement limité par la connaissance de clés d'authentification. Toutefois, le chargement d'une applet malicieuse sur une plateforme ouverte reste envisageable, dans l'hypothèse de l'obtention illégitime des clés de chargement ou par la compromission des commandes d'administration.

Cette expérimentation se place dans le contexte d'une expertise sécuritaire du produit. Dans ce cadre, l'attaque a été réalisée en temps contraint (15 jours) et sur un nombre limité d'échantillons (20 échantillons). Par ailleurs, le code de l'applet comme de la plateforme était disponible à l'expert, de même que les clés permettant le chargement d'applets sur la carte.

La suite de cet article s'articule ainsi : dans un premier temps nous présentons les contres-mesures protégeant le produit qui constitue la cible de notre attaque, et la vulnérabilité détectée. Puis nous présentons l'exploitation de cette vulnérabilité et la compromission de l'applet embarquée sur le produit. Enfin nous concluons en présentant les contre-mesures qui auraient permis de se protéger contre cette attaque.

1. Pour des raisons de confidentialité, les extraits de code présentés dans cet article ont été volontairement modifiés afin de ne pas permettre l'identification du produit concerné.

1 Contre-mesures et vulnérabilité

1.1 Une forteresse bien gardée

La plateforme qui constitue la cible de notre attaque implémente de nombreuses contre-mesures matérielles et logiques. En ce qui concerne les contre-mesures matérielles, toutes les recommandations imposées par le fondeur de la puce sont respectées, ce qui rend une attaque matérielle par injection de faute aussi bien que par canaux cachés difficilement réalisable. En particulier, les détecteurs d'intrusion sont activés, ainsi que tous les mécanismes de désynchronisation matériels. La puce offre un service de chiffrement des données en mémoire volatile et non volatile. Enfin, le module de protection mémoire (MPU) est configuré pour partitionner la mémoire en trois domaines de protection : le code du système d'exploitation, le code des applications natives et le code Java Card. Outre ces contre-mesures purement matérielles, le code de l'applet embarqué est conçu pour détecter les modifications du flot de contrôle.

Les attaques logiques sur les plateformes Java Card reposent pour la plupart sur le mécanisme de "confusion de type". On peut distinguer deux étapes dans les attaques par confusion de type : la création et l'exploitation. La création de la confusion de type consiste à forger une référence d'un type donné vers un objet de type différent [17,10]. L'exploitation de la confusion consiste à accéder à un objet au travers d'une référence de type invalide afin de réaliser des opérations interdites par la machine virtuelle, par exemple accéder à la mémoire au-delà des bornes de l'objet. L'exploitation d'une confusion de type peut s'avérer, selon les contre-mesures présentes sur un produit, aussi complexe que l'obtention de la confusion elle-même.

La protection contre les attaques logiques mise en oeuvre sur notre cible d'attaque repose sur plusieurs mécanismes complémentaires : la pile implicitement typée, le firewall, les méta-données sur les objets, la vérification de cohérence sur le bytecode, la journalisation des erreurs et le chiffrement par applet. Ces mécanismes sont décrits dans la suite de ce chapitre.

Pile implicitement typée. La pile de la machine virtuelle de notre produit cible est implémentée comme une liste d'éléments codés sur 4 octets. Les types natifs supportés par la machine virtuelle sont le type *boolean* (codé sur 1 octet), le type *byte* (codé sur 1 octet) et le type *short* (codé sur 2 octets). Les références d'instances gérées par la machine virtuelle sont quant à elles codées sur 4 octets. Il existe de ce fait un typage implicite

de la pile qui distingue les types natifs codés sur 2 octets au maximum, des références qui sont codées sur 4 octets. Toute tentative de forger une référence à partir d'un *short* aboutirait à créer une référence invalide car incomplète, et l'opération inverse consistant à lire une référence comme un type natif entraînerait la perte de 2 octets, rendant l'information inexploitable. La figure 1 illustre ces deux cas.

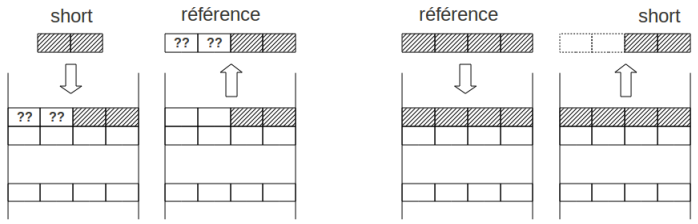

Figure 1. Protection contre la confusion de type par pile typée.

Firewall. Dans le cas où une applet malicieuse parvenait à forger une référence vers une instance appartenant à une autre applet, le mécanisme de firewall imposé par la spécification Java Card empêche tout accès non autorisé entre applets appartenant à des packages différents (hors mécanisme de partage géré par la machine virtuelle). En effet, lors d'une instantiation, la machine virtuelle associe systématiquement chaque instance de classe au package courant via la notion de *contexte*. Lorsqu'un accès aux champs ou méthodes d'une instance est réalisé, la machine virtuelle s'assure que le contexte courant est bien celui associé à l'objet, c'est à dire que l'accès est autorisé.

Méta-données sur les objets. D'un point de vue purement fonctionnel, la représentation d'une instance Java Card en mémoire nécessite d'une part un en-tête indiquant les attributs de l'objet (transient, JCRE entry point, ...) et le contexte de l'objet, d'autre part les données contenues dans les champs de l'objet. Toutefois, afin de se protéger contre les attaques logiques, l'implémentation de machine virtuelle qui constitue notre cible ajoute un ensemble de méta-données dans l'en-tête des objets. Les méta-données additionnelles gérées par la machine virtuelle sont les suivantes :

– Le type de l'objet, c'est à dire l'identifiant unique de classe (assigné par le compilateur Java Card) dont l'objet est instance. Cette infor-

mation est utilisée lors de l'exécution des bytecodes par la machine virtuelle pour vérifier la cohérence entre les types source et destination. Cette contre-mesure permet de se prémunir contre l'exploitation des confusions de type. En effet, lors de l'accès aux champs d'une référence forgée, la machine virtuelle est capable de détecter les incohérences de types entre les champs de la référence originale et les champs de la référence forgée. Il est important de noter que les types natifs (boolean, byte, short) sont indiscernables dans cette représentation,
- Le nombre de champs de l'objet. Ceci permet de détecter les confusions de type entre des classes possédant un nombre de champs différent, visant à accéder à des zones mémoires au delà des zones allouées par la machine virtuelle pour les objets de l'applet.

Vérification de cohérence sur le bytecode. Lors de l'interprétation d'un bytecode, la machine virtuelle vérifie que le type des éléments présents sur la pile est conforme au type des opérandes du bytecode. Par exemple, le bytecode *aload* accepte comme opérande une référence. Lors de l'exécution de ce bytecode, la machine virtuelle vérifie à l'aide des méta-données d'objets que l'élément présent sur la pile correspond bien à une référence. Dans le cas contraire, une erreur est levée par la machine virtuelle.

Journalisation des erreurs. Lorsque la machine virtuelle détecte une situation anormale (par exemple la mise en évidence d'une confusion de type grâce aux contre-mesures présentées plus haut), un compteur d'erreur est incrémenté. Lorsque ce compteur atteint son maximum, la carte est tuée et devient définitivement inutilisable. Cette contre-mesure protège la plateforme en limitant les possibilités d'exploration de l'attaquant. La valeur maximum du compteur est définie en phase de personnalisation des cartes et est fixée à 3 sur nos échantillons.

Chiffrement par applet. Afin d'éviter toute fuite de secret d'une applet, les biens (PIN, clés) des applets sont systématiquement chiffrés par la machine virtuelle avant d'être stockés en mémoire non volatile, avec une clé unique par applet. Ainsi, dans l'hypothèse où un attaquant parvenait à lire la mémoire correspondant aux instances d'une autre applet, ces données seraient inexploitables car indéchiffrables.

L'ensemble de ces mécanismes semble fournir une contre-mesure efficace contre les attaques logiques, en se prémunissant des confusions de

type et en empêchant leur exploitation. La section suivante montre que malgré ces contre-mesures, une vulnérabilité reste présente et exploitable.

1.2 La faille de la cuirasse

Nous avons vu que la pile implicitement typée de la machine virtuelle empêche les confusions de type entre une référence et un type natif. De plus, une confusion de type entre deux références de types différents est rendue impossible par les informations de type et de nombre de champs stockées dans les méta-données des objets. Toutefois, comme nous l'avons vu dans la section 1.1, la confusion de type entre types natifs n'est pas détectée par les contre-mesures de la machine virtuelle. Il est donc possible de convertir de manière illégale une variable de type *byte* en variable de type *short*. Cette attaque n'a qu'un intêret limité car elle permet d'accéder à un unique octet de mémoire de manière illégale.

Un détail de la spécifiation Java Card nous permet d'exploiter plus avant cette vulnérabilité. La spécification Java Card indique que les tableaux, quel que soit le type des références stockées, sont représentés comme des instances de la classe *Object*. Il est donc possible de convertir de manière illégale un tableau de type *byte* en un tableau de type *short*. Les contre-mesures présentées dans la section 1.1 ne permettent pas de prévenir cette attaque. En effet, les méta-données sur les objets ne distinguent pas les types natifs entre eux, ce qui empêche la vérification de cohérence au niveau de l'exécution de bytecode.

Nous pouvons ainsi forger une référence de type tableau de *short* pointant vers un objet de type tableau de *byte*. La lecture des champs de ce tableau de *short* permet d'accéder au contenu de la mémoire volatile de manière illégale. L'accès à la zone mémoire illégale via la confusion de type entre un tableau de *byte* et un tableau de *short* est illustré dans la figure2.

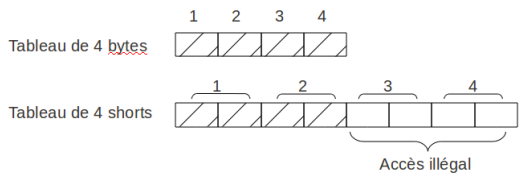

Figure 2. Confusion de type entre un tableau de *byte* et un tableau de short.

Il est important de noter que la confusion de type entre le tableau de type *byte* et le tableau de type *short* a été réalisée en modifiant directement l'archive *cap*, afin de valider le chemin d'attaque. Le fichier résultant est une archive *cap* invalide, c'est à dire que la conversion de type illégale est détectée par le vérificateur de bytecode. La vérification de bytecode, jusqu'à la version 2.2.2 de Java Card, est faite en dehors de la carte. Le chargement de notre archive cap illégale et son exécution sur la carte à puce sont donc autorisés par la machine virtuelle. De plus, les récentes attaques dite "combinées" [3,24,4] permettent de concevoir des applets en apparence légales, qui sont considérées comme valides par le vérificateur de bytecode, et dont le code malicieux est activé une fois chargé sur la carte à l'aide d'une injection de faute. La confusion de type utilisée pour notre attaque peut donc être obtenue via une applet légitime rendue offensive par faute laser.

2 Exploitation

2.1 De Charybde en Scylla

En nous basant sur le mécanisme d'attaque introduit par J. Hogenboom and W. Mostowski [8], nous utilisons les octets lus de manière illégale pour lire l'intégralité de la mémoire non volatile. Le principe de cette attaque, illustré dans la figure 3, est le suivant :

- Dans un premier temps, nous allouons deux tableaux de *byte* successivement en mémoire non volatile,
- Nous créons une confusion de type afin d'accéder au premier tableau de type *byte* comme un tableau de type *short*,
- Les octets accédés de manière illégale offrent un accès en lecture et en écriture à l'en-tête du deuxième tableau,
- Nous modifions la taille (codée sur 2 octets) dans l'en-tête du deuxième tableau, afin de modifier le nombre d'octets accessibles via ce tableau,
- Le deuxième tableau donne accès en lecture et en écriture à 32767 octets de mémoire non volatile (taille maximum d'un tableau autorisée par la machine virtuelle).

L'analyse des octets lus grâce à notre applet montre plusieurs points intéressants. Tout d'abord, l'identifiant d'applet (AID, Application IDentifier) de l'applet bancaire est présent dans les données extraites.

De plus, l'analyse des octets suivants montre qu'ils constituent une suite valide de bytecodes Java Card. Nous avons donc la confirmation que

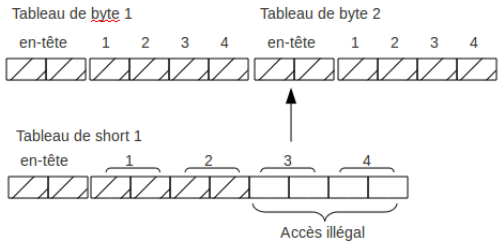

Figure 3. Accès à un en-tête de tableau par confusion de type.

les octets qui ont été lus sur la carte à puce constituent la représentation mémoire de l'applet bancaire déjà présente sur la carte.

2.2 Le loup dans la bergerie

Grâce à une applet malicieuse chargée sur la carte à puce Java Card, nous pouvons accéder en lecture et en écriture aux bytecodes de l'applet bancaire stockés en mémoire non volatile. Nous utilisons cet accès pour compromettre l'applet bancaire de deux manières. La première attaque consiste à transformer la carte en "Yes Card", c'est à dire une carte bancaire qui accepte toute valeur de PIN comme valide. La seconde attaque consiste à extraire les clés secrètes de la carte, ce qui permet de forger de fausses cartes bancaires.

Réalisation d'une "Yes Card" La vérification du PIN dans une transaction bancaire est réalisée au moyen de la commande VERIFY. Au niveau Java Card, l'API fournit un objet PIN possédant une méthode *check* (c.f. listing 1.1), permettant de mettre à jour le flag interne de validation de l'objet PIN. Ce flag est déterminant car il est utilisé par l'application pour s'assurer que le porteur de la carte est authentifié et que la transaction peut donc se poursuivre. Le résultat de la vérification du PIN est retourné sous la forme d'un booléen.

```
1 boolean check(byte[] pin, short offset, byte length)
2     throws ArrayIndexOutOfBoundsException,
3     NullPointerException
```

Listing 1.1. Prototype de la méthode check de la classe PIN.

L'analyse en rétro-conception du bytecode lu sur la carte nous permet d'identifier la zone correspondant à la commande VERIFY de l'application

bancaire, et plus précisément la zone de vérification du PIN. Les listings 1.2 et 1.3 présentent l'interprétation des octets lus sous forme de bytecode et de code Java Card. Cette analyse nous montre que la méthode *check* de l'objet PIN est effectivement appelée pour réaliser la vérification du PIN utilisateur.

```
1  L3: sload 4;
2      slookupswitch
3      L6 2 90 L5 165 L4;
4  L4: getfield_a_this 0;
5      invokevirtual 18;
6      return;
7  L5: sspush 25536;
8      getfield_a_this 0;
9      invokevirtual 19;
10     sor;
11     invokestatic 20;
12 L6: sspush -8531;
13     invokestatic 20;
14     return;
```

Listing 1.2. Interprétation Bytecode de la vérification du PIN.

```
1  result = OwnerPIN.check(pin,offset,SIZE);
2  ...
3  switch(complexResult) {
4  case TRUE:
5          resetPTC();
6          return;
7      case FALSE:
8          UserException.throwIt((short)(0x63C0|(PTC[0])));
9      default:
10         break;
11 }
12 ISOException.throwIt(Terminate);
```

Listing 1.3. Interprétation Java Card de la vérification du PIN.

Nous cherchons à modifier le bytecode correspondant à la vérification du PIN sans pour autant perturber le comportement global de l'application. Pour ce faire, nous nous intéressons en particulier au bytecode *slookupswitch* (ligne 2 du listing 1.2) qui détermine le comportement de l'application en fonction du résultat de la méthode *check*. Les opérandes de l'instruction slookupswitch sont les suivants :

- L'offset du saut à effectuer dans le cas où aucune correspondance n'est trouvée (offset par défaut)
- Le nombre de couples [valeur, offset]
- Une suite de couples [valeur, offset] qui déterminent les sauts de codes correspondant aux différents cas du switch.

En modifiant les octets du tableau obtenu par confusion de type, nous pouvons modifier le code de l'application bancaire. Afin de valider le PIN de manière inconditionnelle, nous modifions la valeur du saut de code dans le cas d'un échec pour que celui-ci soit le même que pour un cas de succès. Cette modification est présentée dans le tableau 1.

	slookup switch	offset par défaut	nombre de cas	valeur échec	offset échec	valeur succès	offset succès
code original	0x75	0x001F	0x0002	0x005A	0x0013	0x00A5	0x000D
code modifié	0x75	0x001F	0x0002	0x005A	**0x000D**	0x00A5	0x000D

Table 1. Modification du code de vérification du PIN.

Après la modification des octets en mémoire, le code équivalent Java Card est celui présenté dans le listing 1.4. Le comportement de la carte est bien celui recherché, c'est à dire que toute valeur de PIN entrée par l'utilisateur est validée par la carte, ce qui se traduit par le retour d'un status "9000" et la permission de continuer la transaction comme un utilisateur authentifié.

```
1  result = OwnerPIN.check(pin, offset, PIN_SIZE);
2  ...
3  switch(result) {
4      case TRUE:
5      case FALSE:
6          resetPTC();
7          return;
8      default:
9          break;
10 }
11 ISOException.throwIt(Terminate);
```

Listing 1.4. Interprétation Java Card du code modifié.

Accès aux clés secrètes Notre deuxième attaque consiste à accéder aux clés secrètes de l'application. Ces clés secrètes font partie des données les plus sensibles d'une application bancaire. Les clés manipulées par l'application permettent d'assurer l'intégrité (clé SMI), la confidentialité (clé SMC), la génération de cryptogrammes (clé AC) et le chiffrement du PIN (clé PIN). Comme présenté précédemment (section 1.1), ces clés sont stockées chiffrées en mémoire non volatile. Il n'est donc pas possible de

lire ces données directement en mémoire via notre attaque de confusion. Toutefois, il est possible d'y accéder en modifiant le code de l'application bancaire.

Afin d'accéder aux variables de clés manipulées dans le bytecode de l'application bancaire, il est nécessaire de déterminer les identifiants uniques de ces variables. Ces identifiants sont attribués par le convertisseur Java Card et stockés dans le composant Constant Pool de l'archive cap. Les bytecodes qui manipulent des variables de classes font référence à ces identifiants pour indiquer sur quelle variable porte l'opération à réaliser.

Pour déterminer les identifiants des clés, nous étudions le mécanisme de terminaison de la carte. Lorsque le compteur d'erreur décrit dans la section 1.1 atteint son maximum, la carte est rendue définitivement inutilisable. Au cours de cette opération, toutes les clés secrètes sont définitivement effacées de la mémoire non volatile. L'analyse en rétro-conception des bytecodes lus sur la carte nous permet d'identifier le code d'effacement des clés, et de déterminer les identifiants de variables utilisés dans cette applet pour manipuler les clés. L'interprétation des octets lus sous forme de bytecode et de code Java Card est donnée dans les listings 1.5 et 1.6 ; les lignes 1, 3, 5, 7 et 9 du listing 1.5 donnent les identifiants de variables correspondant aux clés secrètes de l'applet.

```
1  getfield_a_this 2;
2  invokeinterface 1 13 0;
3  getfield_a_this 3;
4  invokeinterface 1 13 0;
5  getfield_a_this 4;
6  invokeinterface 1 13 0;
7  getfield_a_this 5;
8  invokeinterface 1 13 0;
9  getfield_a_this 6;
10 invokeinterface 1 13 0;
```

Listing 1.5. Interprétation Bytecode de la méthode d'effacement des clés.

```
1 SMI_K.clearKey();
2 SMC_K.clearKey();
3 AC_K.clearKey();
4 PIN_K.clearKey();
5 ICC_K.clearKey();
```

Listing 1.6. Interprétation Java Card de la méthode d'effacement des clés.

Une fois que les identifiants de variables de clés sont connus, il est nécessaire de modifier le code de l'applet bancaire afin que celle-ci retourne

au terminal les octets de clés contenus dans les objets de type Key. Dans l'API Java Card, la méthode getKey (listing 1.7) de l'interface DESKey permet de récupérer les octets de clé contenus dans un objet Key. Cette méthode accepte un tableau de byte qui, en sortie de la méthode, contient les octets de la clé.

```
1 byte getKey(byte[] keyData, short kOff)
2       throws CryptoException
```

Listing 1.7. Prototype de la méthode getKey de la classe DESKey.

Pour accéder aux octets de clé contenus dans l'objets SMI_K de l'applet, nous devons réaliser les opérations suivantes :

1. invoquer la méthode getKey sur l'objet d'identifiant 2 (clé SMI),
2. copier le tableau dans le buffer d'APDU,
3. retourner les données de réponse de la carte.

Comme pour la réalisation de la "Yes Card", nous cherchons à modifier le bytecode de l'application pour récupérer les clés secrètes sans pour autant perturber le comportement global de l'application. Nous devons donc trouver un enchaînement de bytecodes aussi proche que possible de celui que nous voulons exécuter. Une analyse du code de l'applet nous montre que c'est le code réalisant la commande Get Challenge qui se prête le mieux à notre attaque. La commande Get Challenge permet de générer des données aléatoires dans la carte ; ces données aléatoires sont ensuite retournées comme réponse de la carte. La génération d'aléa au niveau Java Card s'appuie sur la méthode *generateData* de l'objet *RandomData* (listing 1.8). La méthode *generateData*, comme la méthode *getKey* de l'interface *DESKey*, accepte un tableau de *byte* qui, en sortie de la méthode, contient les données utiles. Le tableau de *byte* ainsi obtenu est copié dans le buffer d'apdu et renvoyé comme réponse de la carte (listing 1.9).

```
1 public abstract void generateData(byte[] buffer, short offset, short
      length)
2       throws CryptoException
```

Listing 1.8. Prototype de la méthode generateData de la classe RandomData.

Pour mener à bien notre attaque, nous modifions donc le code de l'applet bancaire présenté dans le listing 1.9 en celui présenté dans le listing 1.11. Les différentes modifications apportées au bytecode sont détaillées ci-dessous :

- L'identifiant de variable qui désigne l'objet de type *RandomData* est remplacé par l'identifiant d'objet de la clé secrète (ligne 1 des listings 1.9 et 1.11),
- Le bytecode *invokevirtual* servant à invoquer une méthode de classe est remplacé par le bytecode *invokeinterface* servant à invoquer une méthode d'interface. (ligne 5 des listings 1.9 et 1.11),
- L'identifiant de méthode qui désigne la méthode *generateData* de la classe *RandomData* est remplacé par l'identifiant qui désigne la méthode *getKey* de l'interface *DESKey* (ligne 5 des listings 1.9 et 1.11). L'identifiant d'une méthode pour une interface est donné par sa position dans l'interface (la méthode *getKey* est la quatrième méthode de l'interface *DESKey*, aussi son identifiant est 0x04),
- Le paramètre excédent dans la méthode *generateData* est supprimé en remplaçant le chargement de la valeur par un bytecode nop (ligne 4 des listings 1.9 et 1.11).

Une modification additionnelle, non détaillée ici, a été ajoutée pour contourner une contre-mesure portant sur la génération d'aléa. Afin de se prémunir contre le rejeu, l'application teste le contenu du buffer d'APDU pour valider qu'il est bien identique aux octets d'aléa générés. Dans le cas contraire, la carte est détruite de manière définitive. Nous neutralisons cette contre-mesure en remplacant le bytecode correspondant à l'invocation de l'exception sécuritaire provoquant la destruction de la carte par une suite de bytecodes 'nop' sans effet.

```
1  getfield_a_this 1;
2  getfield_a_this 7;
3  sconst_0;
4  sload_2;
5  invokevirtual 22;
6  getfield_a_this 7;
7  sconst_0;
8  aload_3;
9  sconst_0;
10 sload_2;
11 invokestatic 23;
```

Listing 1.9. Interprétation Bytecode de la méthode de génération d'aléa.

```
1 random.generateData(tmpbuffer, (short)0, size);
2 Util.arrayCopyNonAtomic(tmpbuffer, (short)0, apduBuffer, (short)0,
      size);
3 apdu.setOutgoingLength(size);
4 apdu.sendBytes((short)0, size);
```

Listing 1.10. Interprétation Java Card de la méthode de génération d'aléa.

```
> getfield_a_this 2;
getfield_a_this 7;
sconst_0;
> nop;
> invokeinterface 4;
getfield_a_this 7;
sconst_0;
aload_3;
sconst_0;
sload_2;
invokestatic 23;
```

Listing 1.11. Interprétation Bytecode de la méthode de génération d'aléa modifiée.

```
SMI_K.getKey(tmpbuffer, (short)0);
Util.arrayCopyNonAtomic(tmpbuffer, (short)0, apduBuffer, (short)0,
    size);
apdu.setOutgoingLength(size);
apdu.sendBytes((short)0, size);
```

Listing 1.12. Interprétation Java Card de la méthode de génération d'aléa modifiée.

Suite à ces modifications, détaillées dans le listing 1.11, lors de l'envoi d'une commande Get Challenge, la carte répond par une série d'octets correspondant aux octets de la clé secrète d'intégrité. Le comportement de la carte restant par ailleurs conforme aux spécifications, la carte reste donc totalement fonctionnelle. Il est intéressant de noter que les contre-mesures de chiffrement des biens par applet sont ici inefficaces car c'est bien l'applet bancaire elle-même qui, au travers du code modifié, accède de manière légitime aux objets protégés.

Les valeurs des autres clés peuvent être obtenues de manière similaire en remplaçant l'identifiant de variable qui désigne l'objet de type *RandomData* (ligne 1 du listing 1.9) par un des identifiants obtenus dans la méthode d'effacement des clés (listing 1.5).

Conclusion et contre-mesures

Nous avons présenté dans cet article une attaque logique sur une plateforme Java Card qui compromet intégralement l'applet bancaire embarquée sur la plateforme. En première analyse, le code du produit (plateforme Java Card et applet) fait apparaître un bon niveau de sécurité et semble convenablement protégé contre les attaques logiques au moyen de nombreuses contre-mesures pertinentes. Toutefois, une vulnérabilité apparemment anodine, exploitée convenablement, permet d'accéder en lecture

et en écriture au code de l'applet bancaire embarquée, de modifier le comportement du produit et d'accéder aux biens primaires de l'application.

Pour se prémunir contre l'attaque présentée dans cet article, plusieurs contre-mesures peuvent être mises en place sur le produit expertisé. La première, et la plus intuitive, consiste à étendre le mécanisme de métadonnées sur les objets afin de prendre en compte les types natifs. Grâce à cette contre-mesure, la plateforme Java Card peut détecter les tentatives de confusion entre type *short* et type *byte* qui constituent la première étape pour accéder au code de l'applet bancaire.

Une seconde contre-mesure exploite un mécanisme de protection matériel offert par la puce. Comme présenté dans la section 1.1, la puce fournit un mécanisme de protection de la mémoire (MPU) qui peut être configuré pour partitionner la mémoire en différents domaines de protection. Pour empêcher les attaques logiques provenant d'applets malicieuses, la segmentation mémoire peut être configurée de manière plus fine en définissant un domaine de protection par applet. Ainsi, l'accès illégal au code de l'application bancaire depuis l'applet malicieuse peut être détecté par la pleteforme matérielle. Ces deux contre-mesures, utilisées séparément ou conjointement, permettent de se protéger efficacement contre l'attaque présentée dans cet article.

L'attaque que nous avons réalisée ne peut être mise en oeuvre que sur une plateforme ouverte, c'est à dire une plateforme qui autorise le chargement des applets en phase d'utilisation. De plus, les clés de chargement de la plateforme doivent être connues de l'attaquant afin que celui-ci puisse charger une applet malicieuse. Suite à la réalisation de cette attaque, des recommandations sécuritaires ont été ajoutés au guide d'utilisation de la plateforme afin de renforcer la sécurité du poduit. Ces recommandations imposent de vérifier systématiquement les applets à l'aide du vérificateur de bytecode, et de n'autoriser le chargement d'applets qu'aux entités de confiance. Cette solution, moins technique que les contre-mesures proposées ci-dessus, permet effectivement de se protéger contre l'attaque présentée ici, sous réserve que les utilisateurs respectent effectivement les recommandations imposées par les guides de la plateforme.

Références

1. Dakshi Agrawal, Bruce Archambeault, Josyula Rao, and Pankaj Rohatgi. The em side-channel(s). In Burton Kaliski, çetin Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 29–45. Springer Berlin / Heidelberg, 2003.

2. Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. 2004.
3. Guillaume Barbu, Hugues Thiebeauld, and Vincent Guerin. Attacks on java card 3.0 combining fault and logical attacks. In Dieter Gollmann, Jean-Louis Lanet, and Julien Iguchi-Cartigny, editors, *Smart Card Research and Advanced Application*, volume 6035 of *Lecture Notes in Computer Science*, pages 148–163. Springer Berlin / Heidelberg, 2010.
4. Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Combined software and hardware attacks on the java card control flow. In Emmanuel Prouff, editor, *Smart Card Research and Advanced Applications*, volume 7079 of *Lecture Notes in Computer Science*, pages 283–296. Springer Berlin / Heidelberg, 2011.
5. Jean-Sébastien Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In çetin Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems*, volume 1717 of *Lecture Notes in Computer Science*, pages 725–725. Springer Berlin / Heidelberg, 1999.
6. Christophe Giraud and Hugues Thiebeauld. A survey on fault attacks. In Jean-Jacques Quisquater, Pierre Paradinas, Yves Deswarte, and Anas El Kalam, editors, *Smart Card Research and Advanced Applications VI*, volume 153 of *IFIP International Federation for Information Processing*, pages 159–176. Springer Boston, 2004.
7. GlobalPlatform, Foster City, USA. *GlobalPlatform Card Specification*, version 2.2 edition, March 2006.
8. Jip Hogenboom and Wojciech Mostowski. Full memory read attack on a java card, 2003.
9. Greg Hoglund and Gary McGraw. *Exploiting Software : How to Break Code*. Addison-Wesley, 2004.
10. Julien Iguchi-Cartigny and Jean-Louis Lanet. Developing a trojan applets in a smart card. *Journal in Computer Virology*, 6 :343–351, 2010.
11. Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. pages 388–397. Springer-Verlag, 1999.
12. Oliver Kömmerling and Markus G. Kuhn. Design principles for tamper-resistant smartcard processors. In *Proceedings of the USENIX Workshop on Smartcard Technology on USENIX Workshop on Smartcard Technology*, pages 2–2, Berkeley, CA, USA, 1999. USENIX Association.
13. Julien Lancia. Un framework de fuzzing pour cartes à puce : application aux protocoles EMV. In *Symposium sur la Sécurité des Technologies de l'Information et de la Communication*, SSTIC, pages 350–368, 2011.
14. David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10*, SSYM'01, pages 14–14, Berkeley, CA, USA, 2001. USENIX Association.
15. Thomas S. Messerges, Ezzat A. Dabbish, and Robert H. Sloan. Examining smartcard security under the threat of power analysis attacks. *IEEE Trans. Comput.*, 51 :541–552, May 2002.
16. Simon Moore, Ross Anderson, Paul Cunningham, Robert Mullins, and George Taylor. Improving smart card security using self-timed circuits. In *Technology, Fourth AciD-WG Workshop, Grenoble, ISBN*, pages 211–218, 2002.

17. Wojciech Mostowski and Erik Poll. Malicious code on java card smartcards : Attacks and countermeasures. In Gilles Grimaud and François-Xavier Standaert, editors, *Smart Card Research and Advanced Applications*, volume 5189 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin / Heidelberg, 2008.
18. Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (ema) : Measures and counter-measures for smart cards. In Isabelle Attali and Thomas Jensen, editors, *Smart Card Programming and Security*, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210. Springer Berlin / Heidelberg, 2001.
19. Sergei Skorobogatov and Ross Anderson. Optical fault induction attacks. In Burton Kaliski, çetin Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 31–48. Springer Berlin / Heidelberg, 2003.
20. Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card Platform Security*, 2001. Technical White Paper.
21. Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.2 Application Programming Interface (API)*, 2002.
22. Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.2 Runtime Environment (JCRE) Specification*, 2002.
23. Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.2 Virtual Machine Specification*, 2002.
24. Eric Vetillard and Anthony Ferrari. Combined attacks and countermeasures. In Dieter Gollmann, Jean-Louis Lanet, and Julien Iguchi-Cartigny, editors, *Smart Card Research and Advanced Application*, volume 6035 of *Lecture Notes in Computer Science*, pages 133–147. Springer Berlin / Heidelberg, 2010.