Méthode d'Analyse de Vulnérabilité Appliquée à un Composant de Sécurité d'une Carte à Puce

Samiya Hamadouche[1], Jean-Louis Lanet[2], Mohamed Mezghiche[1]

[1] UMBB/FS/LIMOSE,
5 Avenue de l'indépendance, 35000 Boumerdes, Algérie
{hamadouche-samiya, mohamed-mezghiche}@umbb.dz

[2] XLIM/DMI/SSD,
87 rue d'Isles, 87000 Limoges, France
jean-louis.lanet@unilim.fr
http://www.msi.unilim.fr/~lanet/public/jll.php

Résumé. La sécurité de Java Card repose sur le bon fonctionnement de plusieurs composants opérant de façon complémentaire. Le développement de tels composants, qui se base sur la spécification fournie par Oracle, s'avère très délicat vu la complexité de ces derniers. Il devient alors primordial de s'assurer qu'ils sont bien implémentés, car toute erreur à ce niveau peut représenter un risque de sécurité qui éventuellement aboutira à une attaque. Dans ce papier, nous présentons une approche de génération de suites de test, à partir d'un modèle formel, pour une analyse de vulnérabilité d'un composant de sécurité de la machine virtuelle Java Card : le vérifieur de byte code.

Mots-clés: Java Card, vérifieur de byte code, analyse de vulnérabilité, test à partir de modèles

Abstract. Java Card security relies on several components which operate in a complementary way. The development of such components, based on Oracle specification, is very delicate due to their complexity. So, it is essential to verify the correctness of these components. Indeed, any mistake might provide a security flaw which may lead to an attack. In this paper, we explain an approach to generate test suites from a formal model for a vulnerability analysis of a Java Card Virtual Machine security component: the byte code verifier.

Keywords: Java Card, byte code verifier, vulnerability analysis, model-based testing

1 Introduction

Les cartes à puce sont considérées comme étant des supports d'exécution d'application et de stockage d'informations très sécurisés. Par conséquent, elles doivent faire face aux accès malveillants des attaquants en assurant leur propre sécurité par l'implémentation de plusieurs mécanismes de défense. Cependant, toute erreur d'implémentation de ces derniers peut être exploitée et éventuellement mener à une défaillance du système ou encore révéler des données sensibles de la carte.

Garantir la conformité du système n'est pas toujours suffisante et il devient donc nécessaire de détecter ses vulnérabilités en vue d'améliorer le niveau de sécurité. Nous avons donc développé une méthode de génération de test de vulnérabilité à partir de modèles formels que nous appliquons sur un composant du système d'exploitation d'une carte à puce : le vérifieur de byte code. Ce composant joue un rôle majeur pour la sécurité des cartes à puce de type Java Card et de récentes attaques [7] ont montré que des choix d'implémentation pouvaient ouvrir des failles qu'un attaquant peut utiliser pour en prendre le contrôle. Tester ce composant nécessite de construire des cas de test complexes puisque chacun correspond à une unité de chargement pour une carte dans laquelle nous choisissons d'évaluer chaque champ. La manipulation de cette structure a donc demandé un travail préliminaire d'outillage afin de pouvoir appliquer les transformations requises sur la structure de test. Dans la suite du papier, nous commençons par présenter l'architecture de sécurité de Java Card, en mettant l'accent sur le vérifieur de byte code. Nous exposons par la suite notre approche d'analyse de vulnérabilité appliquée à ce composant de sécurité.

2 Sécurité de la Plateforme Java Card

Java Card est une plateforme, basée sur la technologie Java, destinée au développement d'applications pour cartes à puce (dites *applets*). La machine virtuelle Java Card est le composant responsable de l'interprétation du code des applications chargées dans la carte. L'environnement Java Card est dit ouvert (chargement d'applications autorisé en *post-issuance*) et muli-applicatif (plusieurs applications résident dans la même carte). Cependant, ceci accroit le risque des attaques qui peuvent être menées sur la carte car il devient difficile de garantir l'innocuité du code chargé.

Pour contrer ces attaques, la sécurité de la plate forme Java Card est assurée par plusieurs éléments agissant de façon complémentaire. Avant le chargement, le vérifieur de byte code est le processus offensif de sécurité de Java Card. Il garantit la conformité du code à charger dans la carte par rapport à la spécification de la machine virtuelle. Lors du chargement, la couche logicielle Global Platform assure l'authentification et l'intégrité des applets chargées. Une fois les applets sur la carte, le pare-feu se charge d'isoler les contextes de sécurité afin d'interdire à une applet d'accéder aux données, qui peuvent être relativement sensibles, d'une autre applet.

2.1 Le Vérifieur de Byte Code

C'est un élément crucial de la sécurité de Java Card. Il effectue des vérifications statiques du fichier à charger dans la carte afin de s'assurer qu'il respecte la spécification de la machine virtuelle et donc peut être exécuté sans risques. Il comprend deux parties distinctes mais complémentaires : un vérifieur de structure et un vérifieur de type. Avant de présenter ces deux entités, nous abordons la notion du fichier CAP constituant l'entrée du vérifieur de byte code (dans ce présent travail on s'intéresse aux plates formes Java Card 3.0 Classic Edition).

Le Fichier CAP (Converted APplet). C'est le résultat de la conversion du fichier *Class* et qui sera chargé dans la carte car il représente un format plus simple à exécuter pour une plateforme disposant de peu de ressources. Le format du fichier CAP repose sur la notion de composants. Il est spécifié dans [9] comme étant constitué de 12 composants standards : *Header, Directory, Import, Applet, Class, Method, Static Field, Export, Constant Pool, Reference Location, Descriptor* et *Debug*. Chacun d'entre eux contient des informations spécifiques obtenues à partir du package définissant le fichier CAP correspondant. De plus, les composants possèdent des liens d'interdépendances entre eux.

Les Deux Parties du Vérifieur de Byte Code.

- *Le vérifieur de structure.* Il assure la première étape du processus de vérification. Il a pour objectif de s'assurer que le fichier à charger (i.e. le fichier CAP) soit un fichier bien formé en vérifiant si les informations qu'il contient sont cohérentes. Indépendamment des vérifications purement structurelles, d'autres tests internes plus complexes liés au contenu du fichier sont exécutés pour vérifier qu'il n'y a pas de cycles dans la hiérarchie d'héritage, ou qu'il n'existe pas de méthodes déclarées finales qui soient surchargées. Ce vérifieur effectue deux types de tests : des tests internes suivis d'autres dits externes. Les premiers consistent à s'assurer de la cohérence interne de chaque composant, i.e. vérifier que les données, qui sont sensées représenter un composant, respectent la spécification de Java Card. Les seconds consistent à vérifier les composants entre eux pour s'assurer de la cohérence de l'information partagée.

- *Le vérifieur de type.* Il s'assure qu'aucune conversion de type non autorisée, d'après les règles de typage du langage Java Card, ne soit effectuée par le code à charger. La vérification de type est à la fois la partie la plus complexe du processus de vérification et la plus coûteuse en temps et en mémoire. Ceci vient du fait qu'il faut calculer le type de chaque variable locale et de chaque élément dans la pile d'exécution pour chaque instruction et chaque chemin d'exécution possible.

2.2 Attaques sur les Cartes à Puce

Les attaques actuelles contre les cartes à puce sont essentiellement des attaques matérielles qui nécessitent généralement des moyens physiques et des outils adaptés (lasers, oscilloscopes, etc.) réduisant par leur sophistication le nombre d'attaquants.
Par contre, de nouvelles attaques, dite logiques, commencent à voir le jour et reposent souvent sur la découverte d'une faille et de son exploitation. Ce type d'attaques ne nécessite aucun matériel mais plus difficile à réaliser, est donc à la portée d'un plus grand nombre d'attaquants.

W. Mostowski et E. Poll [7] ont proposé plusieurs attaques sur Java Card en utilisant un code mal typé. L'idée est d'exploiter une confusion de type entre les tableaux de type primitif différents. Il est possible de lire ou d'écrire dans des tableaux de type différent en utilisant une faille dans le mécanisme de transaction de Java Card. J. Iguchi-Cartigny et JL. Lanet ont montré dans [6] qu'un fichier CAP mal formé pourrait conduire à introduire un virus dans une Java Card. Cela est dû à une mauvaise vérification du composant *Reference Location* du fichier CAP. La non cohérence de ce composant permet d'éviter la phase d'édition de lien et de fixer arbitrairement la lecture ou l'écriture d'une adresse dans la mémoire. Il est ainsi possible de réaliser une lecture complète du plan mémoire, tant EEPROM que ROM.

Cependant, nous pensons fortement que d'autres faiblesses sont présentes lors de la vérification du byte code. Dans le cadre de ce papier, nous ne nous focaliserons que sur le vérifieur de structure par lequel la plupart des fautes et fuites d'information peuvent intervenir en créant une brèche de sécurité. Le travail à réaliser sur le vérifieur de type étant différent fait l'objet d'un autre travail de thèse.

3 Approche Proposée

Notre méthode est basée sur une modélisation formelle du vérifieur de structure. Cependant, contrairement à [4] nous utiliserons le modèle pour générer des cas de tests et non pour prouver la correction de l'implémentation du vérifieur. L'un des principaux avantages de l'utilisation d'un modèle formel est de pouvoir raisonner sur ce dernier mais aussi de connaître son comportement très précisément et sans ambiguïté i.e. déterminer les comportements autorisés et ceux refusés. Ce qui permet d'améliorer la qualité et la pertinence des jeux de test.

Plusieurs travaux ont été menés sur la génération automatique de tests à partir de modèles ensemblistes et notamment les travaux de Bull cités par J. Dick et A. Faivre [5]. D'autres travaux utilisent des objectifs de test pour des modèles comme S. Behnia [2] mais très peu de travaux traitent des tests dédiés à la sécurité en particulier sur la négation des spécifications.

Dans le cadre de nos recherches, nous nous intéressons plus particulièrement aux tests de sécurité qui ont pour objectif de s'assurer qu'un comportement rejeté par le modèle l'est aussi par son implémentation. Pour cela, il va donc falloir trouver un moyen de contredire la spécification sans générer l'ensemble de tous les cas possibles, rendant l'exécution de la campagne de test coûteuse, voire impossible. Dans notre cas,

les méthodes formelles offrent de nombreux outils permettant d'extraire des informations très importantes pour la génération de tests.

Les différentes étapes de notre approche sont résumées dans le schéma ci-dessous (Fig.1).

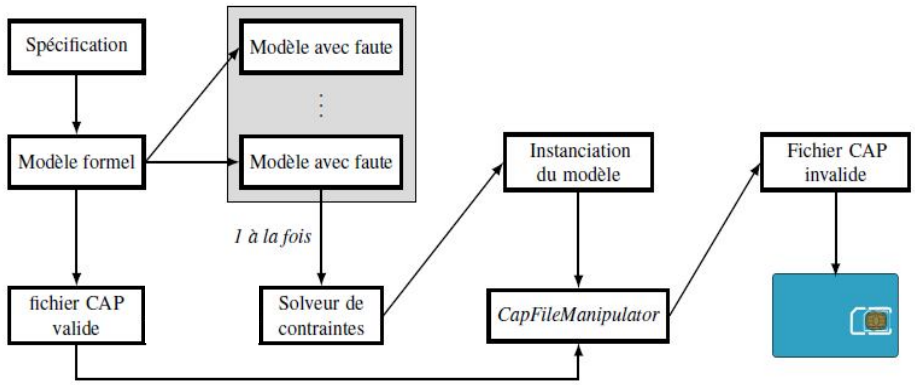

Fig. 1. Plan général de génération de cas de test

Pour illustrer ce processus, nous avons pris comme exemple le composant *Header* dont la spécification informelle est donnée dans [9].

3.1 Construction du Modèle Abstrait du Vérifieur

Partant de la spécification de la machine virtuelle Java Card, chaque composant du fichier CAP est modélisé, suivant le formalisme de la méthode B [1], sous forme d'une machine abstraite comme le montre l'exemple ci-dessous pour le composant *Header*. Les champs de chaque composant ainsi que leurs types sont représentés comme étant des constantes et les contraintes, définies dans la spécification de la machine virtuelle Java Card, comme étant des prédicats.

```
MACHINE
    Header
CONCRETE_CONSTANTS
    /* Types de base */
    t_u1, t_u2, ...
    /* Champs du composant */
    hdr_tag, hdr_size, hdr_magic, hdr_minor_version ...
PROPERTIES
    /* Contraintes */
    hdr_tag : t_u1 &
    hdr_size : t_u2 &
    ...
    hdr_pkg_aid : t_u1 +-> t_u1 &
```

```
    hdr_tag= 1 &
    hdr_size=10+hdr_pkg_aid_length+hdr_pkg_name_length&
    hdr_magic= DECAFFED &
    hdr_major_version : 1 .. 3 &
    hdr_minor_version : 0 .. 2 &
    hdr_flags : 0 .. 7 &
    hdr_pkg_aid_length : 5 .. 16 &
    hdr_pkg_aid : 0 .. hdr_pkg_aid_length-1 --> t_u1&
    ...
END
```

3.2 Dérivation des Modèles avec Fautes

Afin d'avoir des modèles avec faute on a procédé à la négation successive des contraintes du modèle abstrait précédemment obtenu : une contrainte niée à la fois pour obtenir un modèle avec une faute.

Prenons la constante *flags* du modèle précédent. Elle est soumise à une contrainte interne qui indique qu'elle doit appartenir à l'ensemble {0..7}, donc les cas où sa valeur n'appartient pas à cet ensemble seront refusés par le modèle et précisément ce sont ces cas là qui nous intéressent. Dans notre exemple, pour avoir ces cas il faut remplacer cette contrainte par sa négation dans le modèle obtenu à l'étape 3.1. Ceci nous permet d'en déduire que dans le nouveau modèle les valeurs possibles pour *flags* appartiennent à l'ensemble {8 .. t_u1_max}. Ce qui est faux au regard de la spécification. Donc nous n'avons maintenant que des modèles qui permettront d'avoir des fichiers CAP invalides, ce qui correspond à notre objectif.

3.3 Extraction des Cas de Test Abstraits

Chacun des modèles avec faute obtenus est transmis à un solveur de contraintes qui va instancier les variables d'état en respectant toutes les contraintes, y compris celle qui a été modifiée pour introduire l'erreur. Ces instanciations de modèles constituent les cas de test abstraits.

3.4 Génération des Cas de Test Concrets : Les Fichiers CAP Invalides

Les cas de tests abstraits, i.e. les instanciations des modèles avec faute, doivent être transformés en fichiers d'entrée compréhensibles par le vérifieur de byte code : des fichiers CAP. Le *CapFileManipulator* (CFM) [3], une librairie Java permettant de faire la modification et la reconstruction de fichiers CAP, a été développée pour être suffisamment flexible dans l'interprétation des instanciations de modèles. Dans notre cas, pour générer le fichier CAP avec la faute, le CFM prend deux éléments en entrée : un fichier CAP valide et un fichier d'instanciation du modèle. Il va utiliser les valeurs présentes dans l'instanciation du modèle pour modifier les champs correspondants dans le fichier CAP valide.

3.5 Analyse des Vulnérabilités

Le CFM peut être interfacé avec OPAL [8], qui est une librairie Java permettant de faire la gestion des cartes à puce Java Card, en vue de charger des suites de tests dans la carte. Ainsi, nous pouvons envoyer successivement les différents fichiers CAP obtenus au vérifieur et voir sa réaction, ce qui constitue l'oracle. L'acceptation d'un cas de test revient à dire qu'une vulnérabilité est détectée pour le vérifieur ; et comme chacun de ces cas est relatif à une contrainte bien précise, on peut déterminer exactement la contrainte concernée par la vulnérabilité détectée.

4 Evaluation Préliminaire de l'Approche

Après avoir débuté une modélisation conséquente de la description des contraintes pour représenter le vérifieur de structure, nous avons commencé à appliquer notre approche sur le modèle obtenu. Bien que nous n'ayons pas encore assez de résultats expérimentaux, on peut déjà voir l'apport de notre approche par rapport aux plans de tests manuels.

- *Introduction des contraintes.* Nous constatons que la non prise en compte des contraintes revient à une stratégie de fuzzing i.e. la génération de toutes les valeurs possibles pouvant être envoyées à la carte. Mais le problème avec une telle approche est le nombre de cas de tests à générer. En outre, nous obtenons des fichiers acceptables par le vérifieur et d'autres qui ne le sont pas. Donc cette suite de tests ne permet pas de tester uniquement les problèmes de sécurité.
  Ce qui est intéressant avec les contraintes est le fait qu'elles nous permettent de réduire considérablement le nombre de cas de tests à générer car elles limitent les valeurs que peuvent prendre les constantes du modèle. Cependant, il reste encore beaucoup de cas de test. L'utilisation de la technique de test aux bornes, i.e. tester les éléments extrêmes des ensembles, permet de réduire encore ce nombre.
- *La négation des contraintes.* Le fait d'avoir une contrainte qui est fausse dans le modèle nous permet de ne générer que des fichiers CAP invalides au regard de la spécification, ce qui correspond à notre objectif. Par conséquent l'oracle n'aura plus qu'à attendre que le vérifieur de structure indique que le fichier chargé est valide pour conclure sur la présence d'une vulnérabilité.

- *Modèles pré-remplis.* Dans la plupart des cas, la négation d'une contrainte correspond à un nombre relativement restreint de variables à affecter. Dans cette idée, nous pouvons partir de fichiers valides pré-remplis et n'apporter de modification qu'aux variables concernées par la contradiction d'une contrainte. Cette dernière voie de recherche est en cours d'étude. Par conséquent nous n'avons pas encore des résultats suffisants, mais ceux déjà obtenus sont porteurs.

5 Conclusion

Nous avons présenté notre travail sur une méthode d'analyse de vulnérabilité appliquée à un vérifieur de structure de la machine virtuelle Java Card. Partant de la spécification de cette dernière, nous avons construit un modèle abstrait du vérifieur. Ce modèle a été dérivé en appliquant successivement la négation de ses contraintes pour obtenir des modèles avec faute. Par la suite, ces cas de tests abstraits sont instanciés, à l'aide d'un solveur de contraintes. Enfin, nos librairies [3] et [8] permettent d'obtenir les fichiers CAP invalides et d'automatiser le chargement et le déchargement de ces fichiers dans la carte à puce. Cette proposition de méthode d'analyse demande à être validée par des résultats qui consisteront en une analyse qualitative et quantitative des tests générés par rapport à un plan de test manuel. Ce travail sera complété par une analyse du vérifieur de type qui est plus complexe et nécessite de mettre le système dans l'état désiré avant l'exécution du test.

Références

1. Abrial, J. R.: The B Book, Assigning programs to meanings, Cambridge University Press (1996).
2. Behnia, S. : Test de modèles formels en B : cadre théorique et critères de couvertures. Thèse de doctorat, Institut National Polytechnique de Toulouse (2000).
3. CapFileManipulator, http://secinfo.msi.unilim.fr/ softwares (2010)
4. Deville, D., Casset, L., Lanet, J.L.: On Card byte code verification, the ultimate step. JavaOne (2002)
5. Dick, J., Faivre, A.: Automating the generation and sequencing of test cases from model-based specifications. FME'93: Industrial-Strength Formal Methods, pages 268–284. Springer (1993).
6. Iguchi-Cartigny, J., Lanet, J.L.: Developing a Trojan applet in a smart card. Journal in Computer Virology, 1--9 (2009)
7. Mostowski, W., Poll, E.: Malicious code on Java Card smartcards: Attacks and countermeasures. Smart Card Research and Advanced Applications, 1--16 (2008)
8. OPAL, http://secinfo.msi.unilim.fr/softwares (2010)
9. Sun Microsystems. Virtual Machine Specification for the Java Card Platform, Classic Edition (2009)