

République Algérienne Démocratique et populaire
Ministère de l'enseignement supérieur et de la recherche scientifique
Université El-Hadj Lakhdar de Batna
Faculté des sciences
Département Informatique



Mémoire

En vue de l'obtention du diplôme de

Magistère en Informatique

Option : Systèmes Informatiques de Communication (SIC)

Thème :

Vérification de code pour plates-formes embarqués

Présenté par :

Boumassata Meriem

Encadré par :

Pr. Mohamed Benmohammed

Devant le jury

Président : Pr. BILAMI Azeddine, Professeur à l'Université de Batna.

Rapporteur : Pr. BENMOHAMMED Mohamed, Professeur à l'Université de Constantine.

Examineurs : Dr. ZIDANI Abdelmadjid, Maître de Conférences à l'université de Batna.

Pr. CHAOUI Allaoua, Maître de Conférences à l'Université de Constantine.

Année universitaire : 2011-2012

Remerciements

Je remercie tout d'abord monsieur Mohamed Benmohammed, de m'avoir fait l'honneur d'encadrer mon magistère, pour sa disponibilité, et pour son écoute et ses conseils pertinents tout au long de la rédaction de ce mémoire.

Je souhaite remercier l'ensemble des membres de mon jury d'avoir accepté de juger et d'évaluer ce travail de magistère, mais aussi pour avoir passé du temps dans la lecture de ce mémoire.

Je tiens à remercier mes collègues de la promotion du magister, mes enseignants de la première année de magister et les responsables de département d'informatique de l'université de Batna.

Je tiens également à remercier ma famille et mes amis pour leur soutien inconditionnel et leur présence continue.

Enfin je remercie tous ceux qui m'ont aidé de près ou de loin pour la réalisation de ce mémoire.

Boumassata Meriem

*A mes parents,
A mon frère et mes sœurs,
A ma famille,
A mes amis...*

ملخص

البطاقات الذكية أصبحت الآن أجهزة كمبيوتر مصغرة و مبرمجة بلغات كائنية التوجه. قدراتهم الحسابية و التخزينية تسمح لهم باحتواء العديد من التطبيقات. قد تم اليوم، تجهيز الكثير من هذه البطاقات بمجموعة فرعية من منصة جافا (Java). و يطلق على هذه البطاقات اسم بطاقة جافا.

استخدام جافا في مجال البطاقات الذكية يعطي هذه البطاقات القدرة على تحميل برامج قابلة للتنفيذ بعد صدور البطاقة. يتم تحميل البرنامج على البطاقة في شكل محوّل (بايت كود). بعد تحويل هذا البرنامج، يمكن أن يعبر من خلال شبكة للوصول إلى البطاقة. لا شيء يضمن أنه لم يتم تغيير في هذا البرنامج وأنه لا يحتوي على هجوم يمكن أن يهدد أمن البطاقة الذكية كمنصة للتنفيذ.

الآلية الأمنية الرئيسية التي تستخدمها بطاقة جافا للتعامل مع مثل هذه الهجمات هي استعمال متحقق البايت كود الذي توفره شركة SUN Microsystems. في هذه المذكرة، نقترح ونصنع وحدة تحقق من البايت كود و التي يمكنها اكتشاف أخطاء خلط النوع التي لا يكتشفها متحقق البايت كود الموفر من SUN، ونعيد إنتاج متحقق جديد وذلك بإضافة وحدة التحقق التي صنعناها إلى متحقق البايت كود الموفر من SUN.

الكلمات الرئيسية: الأنظمة المدمجة، البطاقات الذكية، بطاقة جافا، الأمن، التحقق، بايت كود.

Résumé

Les cartes à puce sont devenues maintenant des mini-micro-ordinateurs programmables avec des langages orientés objet. Leurs capacités de calcul et de stockage leur permettent de contenir plusieurs applications. Aujourd'hui, une grande quantité de ces cartes sont équipées avec un sous-ensemble de la plate-forme Java. Elles sont appelées Java Cards.

L'utilisation de Java dans les cartes à puce leur apporte la possibilité de charger du code exécutable après la délivrance de la carte. Ce code est chargé sur la carte sous forme de code intermédiaire (bytecode). Après la compilation de ce code, il peut transiter sur un réseau pour arriver à la carte. Rien n'assure que ce code n'a pas été modifié et qu'il ne contient pas une attaque qui peut menacer la sécurité de la carte à puce en tant que plate-forme d'exécution.

Le point clé des mécanismes de sécurité mis en œuvre par Java Card pour faire face à ce type d'attaques est l'utilisation du vérifieur de bytecode fournis par SUN Microsystems. Dans ce mémoire, nous proposons et réalisons un module de vérification de bytecode qui permet de détecter des erreurs de confusion de type qui ne sont pas détectées par le vérifieur de bytecode de SUN. Nous reproduisons un nouveau vérifieur en ajoutons le module de vérification que nous avons réalisé au vérifieur de bytecode de SUN.

Mots clés : Systèmes Embarqués, Cartes à puce, Java Card, Sécurité, Vérification, Bytecode.

Abstract

Smart cards have now become mini-computers programmable with object oriented languages. Their calculation and storage capacities allow them to contain multiple applications. Today, a lot of these cards are equipped with a subset of the Java platform. They are called Java Cards.

The use of Java in smart cards gives them the ability to load executable code after the issuance of the card. This code is loaded onto the card in the form of intermediate code (bytecode). After compiling this code, it can flow through a network to reach the card. Nothing guarantees that this code was not changed and it does not contain an attack that could threaten the security of the smart card as a platform for execution.

The key security mechanism implemented by Java Card to deal with such attacks is the use of the bytecode verifier provided by SUN Microsystems. In this paper, we propose and make a bytecode verification module that can detect type confusion errors that are not detected by the bytecode verifier from SUN, and we reproduce a new verifier by adding the verification module that we have made to the bytecode verifier from SUN.

Keywords: Embedded systems, Smart cards, Java Card, Security, Verification, Bytecode.

Table de matières

Table de matières	i
Liste des figures	viii
Liste des tables	x
Chapitre I : Introduction générale.....	1
1. CONTEXTE.....	2
2. OBJECTIFS	4
3. STRUCTURE DU MEMOIRE.....	5
Chapitre II : La sécurité informatique	7
1. INTRODUCTION.....	8
2. QUELQUES DEFINITONS	8
2.1. Sécurité informatique.....	8
2.2. Menace.....	9
2.3. Attaque.....	9
2.4. Vulnérabilité	9
2.5. Contre-mesure.....	9
2.6. Risque	9
3. PROPRIETES DE LA SECURITE INFORMATIQUE.....	10
3.1. La confidentialité	10
3.2. L'intégrité.....	10
3.3. La disponibilité	10
3.4. L'authentification.....	11
3.5. La non-répudiation.....	11
4. POLITIQUE DE SECURITE.....	11
5. DOMAINES D'APPLICATION DE LA SECURITE INFORMATIQUE.....	12
5.1. Sécurité physique.....	12
5.2. Sécurité de l'exploitation.....	13
5.3. Sécurité logique	13
5.4. Sécurité applicative.....	13
5.5. Sécurité des télécommunications.....	14
6. ATTAQUES LOGIQUES.....	14
6.1. Chevaux de Troie.....	15
6.2. Virus.....	15

6.3.	Vers	16
6.4.	Bombe logique	16
6.5.	Spyware	16
6.6.	Spam	16
6.7.	Hoax.....	17
7.	MECANISMES DE PROTECTION	17
7.1.	L'authentification.....	17
7.2.	Le contrôle d'accès	17
7.3.	Le chiffrement de l'information (cryptographie).....	18
7.3.1.	Quelques définitions	18
7.3.2.	Chiffrement symétrique.....	18
7.3.3.	Chiffrement asymétrique	19
7.3.4.	Fonction de hachage (Signature numérique).....	19
7.4.	Anti-Virus	19
7.5.	Pare-feu (Firewall).....	20
7.6.	Anti-spyware.....	20
7.7.	Autres mesures de protection.....	20
8.	CONCLUSION	20
Chapitre III : La sécurité des systèmes embarqués		21
1.	INTRODUCTION.....	22
2.	LES SYSTEMES EMBARQUES.....	23
2.1.	Définition	23
2.2.	Domaines d'application	23
2.3.	Importance du marché de l'embarqué.....	24
2.4.	Les caractéristiques des systèmes embarqués.....	25
2.4.1.	Fiabilité.....	25
2.4.2.	Faible coût	25
2.4.3.	Limités en ressources.....	25
2.4.4.	Faible consommation.....	26
2.4.5.	Fonctionnement en temps-réel.....	26
2.4.6.	Sûreté de fonctionnement	26
2.4.7.	Sécurité	26
2.5.	Architecture générale d'un système embarqué.....	26
3.	LA SECURITE DANS LES SYSTEMES EMBARQUES	29
3.1.	Exigences de sécurité.....	29

3.2.	Défis de conception d'un système embarqué sécurisé.....	30
3.3.	Attaques sur les systèmes embarqués	31
3.3.1.	Attaques physiques et side-channel.....	31
3.3.2.	Attaques logiques	32
3.4.	La sécurité contre les attaques logiques.....	32
4.	CONCLUSION	33
Chapitre IV : Java Card		35
1.	INTRODUCTION.....	36
2.	LES CARTES A PUCES	37
2.1.	Types de base.....	37
2.2.	Architecture des Smart Cards	38
2.2.1.	Composants physiques	38
2.2.1.1.	CPU.....	39
2.2.1.2.	Mémoires	39
2.2.1.3.	Points de contact	40
2.2.2.	Dispositif d'acceptation de cartes « Card Acceptance Device (CAD) »	40
2.2.3.	Système d'exploitation	41
2.2.4.	Applications.....	41
3.	JAVA CARD.....	42
3.1.	Le langage Java Card	43
3.2.	Architecture d'une Java Card.....	44
3.2.1.	La machine virtuelle Java Card	46
3.2.1.1.	Le convertisseur	47
3.2.1.2.	L'interpréteur	47
3.2.2.	Installeur Java Card et programme d'installation hors-carte	48
3.2.3.	L'environnement d'exécution Java Card.....	49
3.2.3.1.	Cycle de vie d'un JCRE.....	49
3.2.3.2.	Comment fonctionne le JCRE durant une transaction ?	50
3.2.4.	Les APIs Java Card	50
3.2.4.1.	Le package java.lang.....	51
3.2.4.2.	Le package javacard.framework	51
3.2.4.3.	Le package javacard.security	51
3.2.4.4.	Le package javacardx.crypto	51
3.3.	Les Applets Java Card	51
3.3.1.	Identificateur d'application (AID).....	52

3.3.2.	Processus de développement d'une Applet	52
3.3.3.	Installation et exécution d'une applet.....	53
3.3.4.	Structure d'une Applet.....	55
4.	LA SECURITE DANS JAVA CARD	57
4.1.	Les Risques	58
4.1.1.	Motivations.....	58
4.1.2.	Cible d'attaques	59
4.2.	Attaques sur les Java Cards	60
4.2.1.	Attaques physiques	60
4.2.2.	Attaques logiques	61
4.3.	Eléments de sécurité disponibles dans Java Card.....	63
4.3.1.	Sécurité du langage.....	63
4.3.2.	Vérification du sous-ensemble des fichiers classes.....	64
4.3.3.	Vérification du fichier CAP et du fichier d'exportation.....	64
4.3.4.	Chargement d'applet.....	65
4.3.5.	Pare-feu et contextes.....	66
4.3.6.	Points d'entrée et tableaux globaux.....	67
4.3.7.	Partage d'objets	67
4.3.8.	Atomicité des transactions.....	68
4.3.9.	Cryptographie	68
4.4.	Obtenir un code mal typé sur une Java Card	68
4.4.1.	Manipulation du fichier CAP	69
4.4.2.	Abusant des objets d'interface partageable.....	70
4.4.3.	Abuser le mécanisme de transaction	70
5.	CONCLUSION	71
	Chapitre V : Vérification de fichier CAP	72
1.	INTROCUTION.....	73
2.	VUE D'ENSEMBLE DE LA JVM ET DU BYTECODE.....	74
2.1.	Définition du bytecode.....	74
2.2.	Format du fichier class.....	74
2.3.	Bytecode et instructions d'une méthode.....	77
2.4.	Des instructions typées	78
2.5.	Contraintes sur les instructions	79
2.6.	Bytecode et sécurité	81
2.7.	Approche de vérification	81

3.	DIFFERENCE ENTRE JAVA ET JAVA CARD	83
3.1.	Machine virtuelle	83
3.2.	Structure du fichier exécutable	83
3.3.	Vérifieur de bytecode.....	84
4.	VERIFIEUR DE FICHER CAP	86
4.1.	Description du vérifieur	86
4.2.	Processus de vérification	88
4.2.1.	Vérification de la cohérence interne.....	89
4.2.2.	Etapas de la vérification	91
4.2.3.	Vérification du fichier d'exportation.....	92
4.2.4.	Vérification de la compatibilité.....	92
4.2.5.	Considérations sécuritaires	94
5.	CONCLUSION	94

Chapitre VI : Contribution « Introduction d'un nouveau module de vérification dans le vérifieur de fichier CAP » **96**

1.	INTROCUTION.....	97
2.	ENVIRONNEMENT DE DEVELOPPEMENT.....	98
2.1.	Description de l'environnement.....	98
2.2.	Ensemble d'outils	99
2.2.1.	L'outil « jcwde » (Java Card Workstation Development Environment).....	99
2.2.2.	L'outil « cref ».....	100
2.2.3.	L'outil « apdutool »	100
2.2.4.	L'outil « Converter »	100
2.2.5.	L'outil « offcardverifier ».....	101
2.2.6.	L'outil « scriptgen »	102
2.2.7.	L'outil « Installer »	102
2.2.8.	L'outil « capgen ».....	102
2.2.9.	L'outil « capdump »	103
2.2.10.	L'outil « exp2text »	103
2.3.	Ensemble de transformations de fichiers	103
2.4.	Exemple d'utilisation du kit.....	104
2.4.1.	Développement et compilation de l'applet.....	105
2.4.2.	Conversion en fichier CAP.....	106
2.4.3.	Vérification du fichier CAP.....	106
2.4.4.	Installation du fichier CAP	109

2.4.5.	Exécution de l'applet	111
2.4.6.	Utilisation d'une application cliente.....	112
3.	ATTAQUE D'ACCES A LA MEMOIRE COMPLETE SUR UNE JAVA JARD	114
3.1.	Description de l'attaque	114
3.1.1.	Travaux connexes	115
3.1.2.	Confusion de type.....	116
3.1.3.	La vulnérabilité dans le mécanisme de transaction	117
3.1.4.	Réalisation de l'attaque	118
3.1.5.	Exploitation de l'attaque.....	119
3.1.6.	Résultats de l'attaque.....	120
3.2.	Réalisation de l'attaque sur l'applet « MonApplet »	120
3.2.1.	Cible d'attaque.....	120
3.2.2.	Réalisation de l'attaque	121
3.2.3.	Résultats de l'attaque.....	122
4.	CONTRIBUTION.....	124
4.1.	Cause de l'attaque	125
4.2.	Le problème avec les mécanismes de protection	125
4.3.	Solution proposée.....	126
4.4.	Réalisation du nouveau vérifieur	126
4.4.1.	Accéder au code source du vérifieur	127
4.4.2.	Ajouter le nouveau module de vérification	128
4.4.3.	Reproduire le vérifieur.....	130
4.4.4.	Résultats de l'utilisation du nouveau vérifieur	131
5.	DISCUSSION	133
6.	CONCLUSION	135
	Conclusion et perspectives	137
	Bibliographie.....	141
	Annexe : Le concept Java	147
1.	INTRODUCTION.....	148
2.	LE LANGAGE JAVA	148
3.	LA PLATE-FORME JAVA.....	149
3.1.	La machine virtuelle Java	150
3.2.	Les APIs de Java.....	151
4.	CYCLE DE VIE D'UN PROGRAMME JAVA.....	152

4.1.	Structure d'un programme Java	152
4.2.	Exécution d'une application	154
4.3.	Exécution d'une applet	154
4.4.	Exécution d'une servlet	154
5.	LE MODELE DE SECURITE DE JAVA	155
5.1.	Le vérifieur Java	155
5.2.	Le gestionnaire de sécurité.....	156
5.3.	Le chargeur de classe	156
5.4.	Sécurité du langage	157
6.	CONCLUSION	157

Liste des figures

Figure 3.1. Loi empirique de Gordon MOOR pour le processeur Intel.	24
Figure 3.2. Architecture en couches d'un système embarqué.	27
Figure 3.3. Les exigences communes de sécurité des systèmes embarqués.	29
Figure 4.1. Compilation d'un programme Java Card.	44
Figure 4.2. Architecture d'une Java Card.	45
Figure 4.3. Installeur Java Card et programme d'installation hors-carte.	48
Figure 4.4. Processus de développement d'une Applet.	53
Figure 5.1. Structure d'un fichier class.	75
Figure 5.2. Structure de method_info.	76
Figure 5.3. Structure de l'attribut code.	76
Figure 5.4. Exemple de code source Java et le bytecode correspondant.	77
Figure 5.5. Additionner avec une machine à pile.	78
Figure 5.6. Additionner deux entiers avec le bytecode.	79
Figure 5.7. Place du vérifieur dans l'architecture de la JVM.	82
Figure 5.8. L'établissement d'un ensemble de fichiers CAP vérifié.	87
Figure 5.9. Vérification du fichier CAP.	88
Figure 5.10. Vérification de la compatibilité.	93
Figure 6.1. Ensemble de transformations de fichiers pour Java Card.	104
Figure 6.2. Code de l'applet MonApplet.	105
Figure 6.3. Utilisation de l'outil Converter.	106
Figure 6.4. Utilisation de l'outil verifycap.	107
Figure 6.5. Contenu du fichier pme.jca.	108
Figure 6.6. Utilisation de capgen.	108
Figure 6.7. Détection de l'erreur par le vérifieur.	109
Figure 6.8. Utilisation de l'outil scriptgen.	109
Figure 6.9. Utilisation de l'outil cref.	110
Figure 6.10. Utilisation de l'outil apdutool.	111
Figure 6.11. Exécution de l'applet MonApplet.	112
Figure 6.12. Code de l'application Application client.	113
Figure 6.13. Utilisation de l'application cliente.	114
Figure 6.14. La confusion de type dans les tableaux.	116
Figure 6.15. Code de la fonction de débit.	121
Figure 6.16. Exécution de MonAttack avant l'ajout de l'attaque.	122

Figure 6.17. Vérification du fichier packageAttack.cap.....	122
Figure 6.18. L'exception causée par JCSYSTEM.abortTransaction ().....	124
Figure 6.19. Echantillon du code du fichier JCA.	129
Figure 6.20. Projet offcardverifier sous Eclipse.	130
Figure 6.21. Résultat de la vérification avec le nouveau vérifieur.....	132
Figure 6.22. Résultat de la vérification après modification.....	132
Figure 6.23. Résultat de la vérification du fichier pme.cap contenant l'attaque	133

Liste des tables

Tableau 4.1. Eléments supportés et non supportés du langage Java.	43
Tableau 4.2. Description des composants d'architecture d'une Java Card.	46
Tableau 4.3. Identificateur d'application (AID).	52

Chapitre I :

Introduction Générale

1. CONTEXTE

Le mot « sécurité » est très utilisé de nos jours, il s'applique dans des domaines très différents. Les systèmes informatiques constituent un de ces domaines. Ces systèmes sont nombreux (PC, cartes à puce, Internet, etc.) et de plus en plus utilisés.

Les exigences des utilisateurs en termes de sécurité sont de plus en plus importantes. La sécurité informatique a pour but la protection des ressources matérielles et logicielles d'un système informatique contre la révélation, la modification, ou la destruction accidentelle ou malintentionnée. [1]

Afin de pouvoir sécuriser un système, il est nécessaire de mettre en œuvre des mécanismes de protection au sein de ce système. La mise en œuvre de ces mécanismes se justifie par l'existence de menaces pouvant se concrétiser sous forme d'attaques. Il est donc essentiel d'identifier les menaces potentielles portant sur le système, de connaître et de prévoir la façon de procéder de ces menaces, pour pouvoir déterminer les mécanismes de protections qu'on doit mettre en œuvre au sein de ce système.

La menace est caractérisée par la possibilité et la probabilité d'attaque contre la sécurité. Elle engendre des risques et des coûts humains et financiers : perte de confidentialité de données sensibles, indisponibilité des infrastructures et des données, etc. Les attaques exploitent les vulnérabilités du système informatique, chaque système informatique est exposé à une famille d'attaques selon les vulnérabilités qu'il présente. [2] [3]

Notre travail ne porte pas sur la sécurité des systèmes informatiques en général, mais sur les systèmes embarqués, en particulier les cartes à puce. Ces cartes sont de plus en plus utilisées à travers le monde et leurs capacités de calcul et de stockage leur permettent de contenir plusieurs applications. Pour programmer ces applications, divers langages existent, dont celui qui nous intéresse, Java Card.

Java Card est l'adaptation de Java pour les cartes à puce et d'autres dispositifs à mémoire limitée. C'est l'une des dernières plates-formes pour carte à puce, et la plus populaire. Aujourd'hui, une grande quantité de cartes à puces sont équipées avec une plate-forme Java Card. Elles sont largement utilisées dans les cartes SIM (utilisées dans les téléphones mobiles), les cartes de guichets automatiques, et les services financiers (cartes de crédit, porte-monnaie électroniques, etc.).

Les cartes à puces Java Cards sont utilisées pour saisir, stocker, manipuler, et accéder à des données sensibles (codes d'accès aux comptes, informations personnelles, données médicales, etc.) dont la perte peut causer une sérieuse détresse ou de graves préjudices financiers. La sécurité de ces cartes est donc devenue une préoccupation importante. C'est un sujet en cours et peut rapidement devenir un problème encore plus grand que la sécurité des ordinateurs actuels. Ceci est dû aux contraintes liées aux dispositifs matériels (les ressources limitées et les contraintes de puissance) et une variété de vulnérabilités inhérentes rencontrés lors de la mise en œuvre des mesures de sécurité, ou bien au coût de la sécurité. [1][4]

Dans ce travail, nous nous intéressons aux problèmes de sécurité provenant du chargement d'applications Java Card sur les cartes à puces. Parmi les avantages de l'utilisation de Java dans les cartes à puce, on trouve la possibilité de charger de nouvelles applications sur la carte après sa délivrance. Cependant, cette possibilité révèle de grands problèmes de sécurité au niveau de la Java Card. Les applications sont compilées en dehors de la carte à puce et sont chargées sur celle-ci sous forme d'un code intermédiaire (bytecode) contenu dans un fichier qui a un format appelé CAP. Avant son chargement dans la carte, le bytecode peut transiter sur un réseau (par exemple sur Internet) pour arriver à la carte. Rien n'assure que le contenu de ce bytecode n'a pas été modifié et ne contient pas un code malveillant (une attaque) qui peut menacer la sécurité de la carte en tant que plate-forme d'exécution, ainsi que celle des autres applications présentes sur la carte.

Pour faire face à ce problème, SUN Microsystems a mis en œuvre un ensemble de mécanismes de sécurité qui reposent principalement sur la vérification du fichier CAP avant son chargement dans la carte à puce. Ce mécanisme consiste à une vérification statique du bytecode contenu dans le fichier CAP. Il permet d'assurer qu'aucun programme malveillant n'est exécuté par la machine virtuelle. La partie essentielle, et non triviale, de la vérification du bytecode est la vérification des propriétés du typage de bytecode, par exemple, la vérification que les arguments d'une instruction sont toujours des types attendus par cette instruction, ou bien la vérification qu'une méthode d'initialisation d'une classe donnée a été invoquée avant l'utilisation d'une instance de cette classe. La vérification de ces propriétés de manière statique avant l'exécution assure que l'exécution du code par la machine virtuelle Java Card ne risque pas de compromettre l'intégrité de la carte ou la confidentialité des autres données qui y sont stockées. [5]

Le vérifieur de fichier CAP de SUN a été montré très efficace pour la détection des erreurs de typage dans le bytecode de Java Card. Il permet à la carte de rejeter toute application contenant un code malveillant. Cependant, les travaux de [6] ont montré la présence d'une vulnérabilité dans le système Java Card, qui peut être utilisée pour introduire une erreur de confusion de type au niveau du bytecode contenu dans le fichier CAP sans que le vérifieur de fichier CAP la détecte. Ils ont exploité cette confusion de type pour créer une attaque d'accès à la mémoire EEPROM complète d'une Java Card. On peut donc, accéder au code et aux données des autres applications présentes sur la carte et effectuer des actions malveillantes. Par exemple, un attaquant peut exploiter cette attaque pour accéder au code d'une application de porte monnaie électronique contenue dans une Java Card et modifier dans ce code à fin d'obtenir un gain financier.

2. OBJECTIFS

Notre travail consiste à réaliser une contre mesure sécuritaire pour faire face à ce type d'attaques. Nous visons, dans ce mémoire, à atteindre les objectifs suivants :

- Comprendre le fonctionnement des Java Card : le développement des applications, leur chargement sur la carte, et leur exécution.
- Avoir une idée générale sur les mécanismes de sécurité mis en œuvre contre les attaques de confusion de type, au sein des Java Cards.
- Comprendre les détails du fonctionnement du vérifieur statique de fichier CAP fournis par SUN.
- Utiliser un simulateur de l'environnement Java Card pour développer des applications Java Card, exploiter l'attaque de lecture de mémoire complète dans une carte de porte monnaie électronique, et démontrer l'inefficacité du vérifieur de SUN à détecter ce type d'attaques.
- Déterminer la vulnérabilité, au sein du système Java Card, qui permet de réaliser l'attaque, et comprendre les raisons qui permettent à cette attaque de passer le vérifieur de fichier CAP. Ensuite, proposer une solution pour faire face à cette attaque.
- Réaliser un module de vérification de fichier CAP qui permet de détecter ce type d'attaques et l'introduire dans le vérifieur de fichier CAP hors-carte de SUN fournis par le simulateur.

- Utiliser le simulateur de l'environnement Java Card pour démontrer l'efficacité du nouveau vérifieur à détecter ce type d'attaques et comparer les résultats de la vérification par le nouveau vérifieur avec celles qui sont obtenues par l'ancien vérifieur.

3. STRUCTURE DU MEMOIRE

Le mémoire est composé de six chapitres :

- Le premier chapitre est une introduction générale.
- Le deuxième chapitre présente les notions de base et les terminologies liées à la sécurité informatique, ainsi que les attaques qui menacent la sécurité logique d'un système informatique, essentiellement ceux qui parviennent par Internet, et les mécanismes utilisés pour protéger le système contre ces attaques.
- Le troisième chapitre présente les systèmes embarqués, leurs caractéristiques, et les composants matériels les plus communes de la plupart de ces systèmes. Il présente, aussi, l'aspect sécurité dans ces systèmes en décrivant leurs exigences de sécurité, les défis de conception d'un système embarqué sécurisé et les attaques possibles sur ces systèmes tout en concentrant sur les attaques logiques et la sécurité contre ces attaques.
- Le quatrième chapitre présente les cartes à puce de type Java Card en décrivant leurs composants matériels et l'utilisation des aspects de Java dans ces cartes (le langage, l'environnement d'exécution, la machine virtuelle). Il décrit le fonctionnement de ces cartes et comment se déroule l'installation d'applications externes sur celles-ci. On présente aussi, dans ce chapitre, l'aspect sécurité dans les Java Cards, les problèmes apportés par le chargement d'applications externes, précisément l'introduction d'un code mal typé, et un état de l'art sur les éléments de sécurité utilisés par Java Card pour traiter ces problèmes.
- Le cinquième chapitre présente le rôle et la place qu'occupe le vérifieur statique de fichier CAP au sein de la machine virtuelle Java Card, et présente le vérifieur statique de fichier CAP hors-carte tel qu'il est décrit par la spécification de Sun, tout en décrivant la façon de procéder de ce vérifieur et les différentes vérifications qu'il effectue.

- Le sixième chapitre présente l'environnement de développement qui contient le simulateur de Java Card, la vulnérabilité présente dans le système Java Card, et l'attaque d'accès à la mémoire EEPROM complète de Java Card, ainsi que son exploitation dans notre projet. Il présente la solution que nous proposons pour faire face à ce type d'attaques, la réalisation du nouveau module de vérification et son introduction au vérifieur fourni par le simulateur, et une discussion des résultats obtenus par la vérification du fichier CAP avec le nouveau vérifieur.

Ce mémoire termine par une conclusion qui résume notre travail et quelques perspectives qui présentent nos remarques concluantes et nos suggestions pour une recherche future.

Chapitre II :

La sécurité informatique

1. INTRODUCTION

Avec l'automatisation des systèmes d'information et l'utilisation d'un nombre important d'ordinateurs pour stocker des informations vitales et sensibles, la nécessité de sécuriser les systèmes d'informations devient plus apparente.

Le problème de la protection des informations sur les ordinateurs est devenu encore plus critique et difficile depuis l'adoption de l'Internet. L'Internet a rendu les ordinateurs à travers le monde interconnectés et malgré les nombreux avantages du partage des données et l'échange d'informations qu'il offre, l'Internet est également devenu la route principale à la pénétration aux systèmes par des utilisateurs non autorisés qui peuvent effectuer des actions malveillantes.

Il est donc essentiel de connaître les ressources du système à protéger et de maîtriser le contrôle d'accès et les droits des utilisateurs du système d'information. [7]

Dans ce chapitre, nous allons introduire les principes de la sécurité informatique et les propriétés d'un système sécurisé. Nous définirons ainsi, brièvement, qu'est ce qu'une politique de sécurité et les domaines d'application de la sécurité informatique. Nous allons nous concentrer sur les attaques qui menacent la sécurité logique d'un système informatique, essentiellement ceux qui parviennent par Internet, et les mécanismes utilisés pour protéger le système contre ces attaques.

2. QUELQUES DEFINITIONS

D'une manière générale le système d'information concerne l'ensemble des moyens (organisation, acteurs, procédures et systèmes informatiques) nécessaires à l'élaboration, au traitement, au stockage, à l'acheminement et à l'exploitation des informations. L'essentiel du système d'information est porté par le système informatique et donc assurer la sécurité de l'information implique d'assurer la sécurité des systèmes informatiques.

Nous présentons dans cette section quelques définitions des notions de base de la sécurité informatique.

2.1. Sécurité informatique

La sécurité informatique est le domaine de l'informatique qui analyse les propriétés de sécurité des systèmes informatiques. Elle a pour but la protection des ressources matérielles et logicielles (incluant les données et les programmes) d'un système informatique contre la révélation, la modification, ou la destruction accidentelle ou malintentionnée.

Afin de pouvoir sécuriser un système, il est nécessaire d'identifier les menaces potentielles portant sur celui-ci, et donc de connaître et de prévoir la façon de procéder de ces menaces et ensuite la mise en œuvre des mécanismes de sécurité qui permettent de minimiser la vulnérabilité d'un système informatique contre ces menaces. [8]

2.2. Menace

La menace (en anglais « threat ») représente le type d'action susceptible de nuire dans l'absolu. Les menaces sont caractérisées par les possibilités et les probabilités d'attaque contre la sécurité. Elles engendrent des risques et des coûts humains et financiers : perte de confidentialité de données sensibles, indisponibilité des infrastructures et des données, etc. Les risques peuvent se réaliser si les systèmes menacés présentent des vulnérabilités. [2] [3]

2.3. Attaque

Une attaque est une action visant à violer une (ou plusieurs) propriétés de sécurité des systèmes informatiques. C'est l'exploitation d'une faille d'un système informatique (système d'exploitation, réseau, logiciel ou bien même de l'utilisateur) à des fins non connues par l'exploitant du système et généralement préjudiciables. [1]

2.4. Vulnérabilité

La vulnérabilité (en anglais « vulnerability », appelée parfois faille ou brèche) est une faute de conception ou de configuration du système informatique, intentionnelle ou accidentelle, qui favorise la réalisation d'une menace ou la réussite d'une attaque.

2.5. Contre-mesure

La contre-mesure est l'ensemble des actions mises en œuvre en prévention de la menace.

2.6. Risque

Les risques sont le résultat de la combinaison des menaces et des vulnérabilités. Ils doivent être évalués, soit pour obtenir le meilleur compromis possible entre sécurité et coût pour un système donné, soit simplement pour calculer le montant des primes d'assurance pour couvrir ces risques.

Le risque en termes de sécurité est généralement caractérisé par l'équation suivante :

$$\text{Risque} = (\text{menace} * \text{vulnérabilité}) / \text{contre-mesure} \quad [2] \quad [7]$$

3. PROPRIETES DE LA SECURITE INFORMATIQUE

La sécurité informatique, d'une manière générale, consiste à assurer que les ressources d'un système d'information sont uniquement utilisées dans le cadre prévu. L'objectif est d'assurer que ces ressources aient les 5 propriétés suivantes : la confidentialité, l'intégrité, la disponibilité, l'authentification et la non-répudiation.

3.1. La confidentialité

« La confidentialité est le maintien du secret des informations » (Le Petit Robert).

Dans le cadre d'un système d'information, la confidentialité est la propriété d'une information de ne pas être disponible ou révélée à des utilisateurs non autorisés à la connaître. Cette propriété garantit que l'information considérée est accessible au moment voulu par les personnes autorisées. Ceci signifie que le système informatique doit :

- Empêcher que des utilisateurs puissent connaître une information confidentielle s'ils n'y sont pas autorisés ;
- Empêcher que des utilisateurs autorisés à connaître une information confidentielle puissent la divulguer à des utilisateurs non autorisés.

3.2. L'intégrité

L'intégrité est la propriété d'une information d'être correcte. Cette propriété garantit que l'information est bien celle que l'on croit être. Cela signifie que le système informatique doit :

- Empêcher une modification de l'information par des utilisateurs non autorisés ou une modification incorrecte par des utilisateurs autorisés ;
- Faire en sorte qu'aucun utilisateur ne puisse empêcher la modification légitime de l'information; par exemple, empêcher la mise à jour périodique d'un compteur de temps serait une atteinte contre l'intégrité.

3.3. La disponibilité

La disponibilité est la propriété d'une information d'être accessible lorsqu'un utilisateur autorisé en a besoin. Cette propriété permet de maintenir le bon fonctionnement du système d'information. Cela signifie que le système informatique doit :

- Fournir l'accès à l'information pour que les utilisateurs autorisés puissent la lire ou la modifier ;
- Faire en sorte qu'aucun utilisateur ne puisse empêcher les utilisateurs autorisés d'accéder à l'information.

La propriété de disponibilité s'applique aussi au service fourni par le système informatique et une atteinte contre la disponibilité d'un service est souvent appelée déni de service.

3.4. L'authentification

- *Identifier* : obtenir l'identité d'une personne ou d'une entité.
- *Authentifier* : vérifier qu'une personne ou une entité correspond bien à son identité déclarée (par exemple par identifiant et mot de passe).
- *Autoriser* : vérifier qu'une personne ou une entité a les droits nécessaires pour accéder ou modifier à une ressource.

L'authentification consiste à assurer l'identité d'un utilisateur, c'est-à-dire de garantir à chacun des correspondants que son partenaire est bien celui qu'il croit être. Un contrôle d'accès peut permettre (par exemple par le moyen d'un mot de passe) l'accès à des ressources uniquement aux personnes autorisées.

3.5. La non-répudiation

La non-répudiation est le fait de ne pouvoir nier ou rejeter qu'un événement a eu lieu. Par exemple empêcher l'émetteur ou le destinataire de nier la transmission ou la réception d'un message. [7] [9]

4. POLITIQUE DE SECURITE

La sécurité des systèmes informatiques se cantonne généralement à garantir les droits d'accès aux données et ressources d'un système en mettant en place des mécanismes d'authentification et de contrôle permettant d'assurer que les utilisateurs des dites ressources possèdent uniquement les droits qui leur ont été octroyés.

Les mécanismes de sécurité mis en place peuvent néanmoins provoquer une gêne au niveau des utilisateurs et les consignes et règles deviennent de plus en plus compliquées au fur et à mesure que le réseau s'étend. Ainsi, la sécurité informatique doit être étudiée de telle manière à ne pas empêcher les utilisateurs de développer les usages qui leur sont nécessaires, et de faire en sorte qu'ils puissent utiliser le système d'information en toute confiance.

C'est la raison pour laquelle il est nécessaire de définir dans un premier temps une **politique de sécurité**, dont la mise en œuvre se fait selon les quatre étapes suivantes :

- Identifier les besoins en terme de sécurité, les risques informatiques pesant sur le système d'information et leurs éventuelles conséquences ;

- Elaborer des règles et des procédures à mettre en œuvre dans les différents services de l'organisation pour les risques identifiés ;
- Surveiller et détecter les vulnérabilités du système d'information et se tenir informé des failles sur les applications et matériels utilisés ;
- Définir les actions à entreprendre et les personnes à contacter en cas de détection d'une menace.

Le but d'une politique de sécurité est de définir les droits d'accès des utilisateurs, les actions autorisées et celles qui ne le sont pas au niveau d'un système d'information. Elle est constituée par l'ensemble des lois, règles et pratiques qui régissent le traitement des informations sensibles et l'utilisation des ressources par le matériel et le logiciel d'un système. [1] [7]

5. DOMAINES D'APPLICATION DE LA SECURITE INFORMATIQUE

La sécurité d'un système informatique est comparée régulièrement à une chaîne en expliquant que le niveau de sécurité d'un système est caractérisé par le niveau de sécurité du maillon le plus faible. Ainsi, une porte blindée est inutile dans un bâtiment si les fenêtres sont ouvertes sur la rue.

Cela signifie que la sécurité doit être abordée dans un contexte global. En effet tous les domaines de l'informatique sont concernés par la sécurité d'un système d'information. En fonction de son domaine d'application, la sécurité informatique se décline en : sécurité physique, sécurité de l'exploitation, sécurité logique, sécurité applicative et sécurité des télécommunications.

5.1. Sécurité physique

Elle concerne tous les aspects liés à l'environnement dans lequel les systèmes se trouvent. C'est la sécurité au niveau des infrastructures matérielles : salles sécurisées, lieux ouverts au public, postes de travail des personnels, alimentation électrique, climatisation, etc.

- **Mesures pour la sécurité physique**
 - Respect de normes de sécurité ;
 - Protection de l'environnement contre les accidents (incendie, température, humidité, ...)
 - Protection des accès physiques ;
 - Application de la redondance physique ;

- Mise en œuvre d'un plan de maintenance préventive (ex. test) et corrective (ex. pièce de rechange), etc.

5.2. Sécurité de l'exploitation

Elle concerne la sensibilisation des utilisateurs aux problèmes de sécurité. Elle vise le bon fonctionnement des systèmes. Cela comprend la mise en place d'outils et de procédures relatifs aux méthodologies d'exploitation, de maintenance, de test, de diagnostic et de mise à jour.

- *Mesures pour la sécurité d'exploitation*
 - Mise en œuvre d'un plan de sauvegarde, de secours, de continuité et de tests ;
 - Application des inventaires réguliers et si possible dynamiques ;
 - Gestion du parc informatique, des configurations et des mises à jour ;
 - Contrôle et suivie de l'exploitation, etc.

5.3. Sécurité logique

Elle concerne la sécurité au niveau des données, notamment les données du système d'information, les applications ou encore les systèmes d'exploitation. Elle fait référence à la réalisation de mécanismes de sécurité par logiciel.

- *Mesures pour la sécurité logique*
 - Mise en œuvre d'un système de contrôle d'accès logique s'appuyant sur un service d'authentification, d'identification et d'autorisation ;
 - Mise en place des dispositifs pour garantir la confidentialité dont la cryptographie ;
 - Gestion efficace des mots de passe et des procédures d'authentification ;
 - Mise en place des mesures antivirus et de sauvegarde d'informations sensibles, etc.

5.4. Sécurité applicative

L'objectif est d'éviter les « bugs » dans les applications.

- *Mesures pour la sécurité applicative*
 - Application d'une méthodologie de développement des applications ;
 - Assurance de la robustesse des applications ;
 - Réalisation de contrôles programmés et des jeux de test ;
 - Mise en œuvre d'un plan de migration d'applications critiques ;
 - Validation et l'audit des programmes ;
 - Mise en œuvre d'un plan d'assurance de sécurité, etc.

5.5. Sécurité des télécommunications

Elle concerne les technologies réseau, les serveurs de l'infrastructure, les réseaux d'accès, etc. Elle permet d'offrir à l'utilisateur final une connectivité fiable et de qualité de « bout en bout ».

- **Mesures pour la sécurité des télécommunications**

La mise en œuvre d'un canal de communication fiable entre les correspondants, quels que soit le nombre et la nature des éléments intermédiaires. Cela implique la réalisation d'une infrastructure réseau sécurisée au niveau des accès, des protocoles de communication, des systèmes d'exploitation et des équipements. [7] [10]

- **Remarque**

Dans ce mémoire, nous nous intéressons qu'à la sécurité logique. Nous n'allons donc présenter que les attaques logiques qui surviennent généralement à travers le réseau Internet, et les mécanismes de protection contre ces attaques.

6. ATTAQUES LOGIQUES

Les motivations des attaques logiques peuvent être de différentes sortes : obtenir un accès au système, voler des informations, tels que des secrets industriels ou des propriétés intellectuelles, glaner des informations personnelles sur un utilisateur, récupérer des données bancaires, s'informer sur l'organisation (ex. entreprise de l'utilisateur), utiliser les ressources du système de l'utilisateur, notamment lorsque le réseau sur lequel il est situé possède une bande passante élevée, etc.

Sur Internet des attaques ont lieu en permanence, à raison de plusieurs attaques par minute sur chaque machine connectée. Afin de contrer ces attaques il est indispensable de connaître les principaux types d'attaques afin de mettre en œuvre des dispositions préventives.

Les attaques les plus connues sont les attaques à travers les codes malicieux.

- **Code malicieux**

Un code malicieux est un fragment de code écrit dans un langage de programmation pouvant, à l'insu de l'utilisateur, porter atteinte à la confidentialité, à l'intégrité, aux flots de contrôle et de données ainsi qu'à la fonctionnalité d'un environnement d'exécution ou sur l'environnement d'exécution au complet en le mettant par exemple hors service (déni de service ou disponibilité).

Les codes malicieux se répandent en général par le réseau, soit par accès direct à l'ordinateur attaqué, soit cachés dans un courriel ou sur un site Web attrayant, mais aussi éventuellement par l'intermédiaire d'une disquette, d'une clé USB ou d'un CD-Rom. La destination de ces logiciels est de s'installer sur l'ordinateur dont ils auront réussi à violer les protections pour y commettre des méfaits, et aussi pour se propager vers d'autres victimes.

Nous introduisons ici quelques-unes des grandes familles de code malicieux.

6.1. Chevaux de Troie

Un cheval de Troie (« trojan » en anglais) est un programme qui, sous les apparences d'un logiciel utile, autorise l'exécution de commandes sur un ordinateur, depuis un autre ordinateur distant, via Internet.

- *Les backdoors*

Certains chevaux de Troie, les backdoors, permettent de contrôler à distance un ordinateur après l'avoir infecté (lors du téléchargement d'un fichier ou l'ouverture d'une pièce jointe), le programme permet, lorsque l'ordinateur est en connexion Internet, d'avoir un accès libre en lecture, écriture ou suppression à la totalité des fichiers présents sur le disque dur de cet ordinateur mais également de faire exécuter à cet ordinateur des actions illégales (attaques de serveurs, intrusions dans des sites sensibles, etc.).

- *Les canaux cachés*

Un attaquant utilisant un Cheval de Troie a généralement pour but soit de détruire de l'information soit simplement d'y accéder. Afin d'obtenir l'accès aux informations sensibles, l'intrus doit trouver un moyen de se faire passer pour un utilisateur autorisé ou bien trouver un chemin caché pour que le Cheval de Troie fasse suivre l'information. Un canal caché est un moyen de transférer de l'information à travers un chemin peu suspect et non surveillé.

C'est un moyen moins direct de communiquer. Par exemple, si deux parties qui souhaitent communiquer partagent une ressource, des données peuvent être transmises en utilisant les changements d'état de cette ressource (ex. la création ou la modification d'un fichier donné).

6.2. Virus

Un virus informatique est un programme conçu pour se dupliquer ; il se propage par tous les moyens d'échange de données numériques (Internet, réseau, disquette, CD-ROM, clé USB, etc.) ; les effets d'un virus sont très variés, de l'affichage d'un simple message anodin à la destruction complète de toutes les données de l'ordinateur.

Le terme virus est réservé aux logiciels qui s'installent sur un ordinateur à l'insu de son utilisateur avec un but malveillant, parce qu'il existe des usages légitimes de cette technique dite de *code mobile* : les appliquettes Java et les procédures JavaScript sont des programmes qui viennent s'exécuter sur un ordinateur en se chargeant à distance depuis un serveur Web visité, sans que toujours on en ait conscience, et en principe avec un motif légitime. Les concepteurs de Java et de JavaScript nous assurent qu'ils ont pris toutes les précautions nécessaires pour que ces programmes ne puissent pas avoir d'effet indésirable sur nos ordinateurs, bien que ces précautions, comme toutes précautions, soient faillibles. [3] [11] [12]

6.3. Vers

Les vers sont des programmes qui se comportent comme des virus. Ils ne diffèrent de ceux-ci que du fait qu'ils ne nécessitent pas de s'incorporer dans un fichier hôte.

6.4. Bombe logique

Une Bombe logique est une partie d'un programme malveillant (virus, cheval de Troie, etc.) qui reste dormante dans le système hôte jusqu'à ce qu'un instant ou un événement survienne, ou encore que certaines conditions soient réunies, pour déclencher des effets dévastateurs en son sein. Le virus Tchernobyl, qui fut l'un des virus les plus destructeurs, avait une bombe logique qui s'est activée le 26 avril 1999. [13]

6.5. Spyware

Un spyware (ou logiciel espion) est un programme conçu pour collecter des données personnelles sur son utilisateur et les envoyer, à son insu, à un tiers via Internet.

Les spywares ne sont pas des virus parce qu'ils ne mettent pas en danger l'intégrité du système, des applications et des données. Mais leurs actions posent des problèmes éthiques et juridiques, quant à la violation de la vie privée.

6.6. Spam

Les spams consistent en « communications électroniques massives, notamment de courrier électronique, sans sollicitation des destinataires, à des fins publicitaires ou malhonnêtes ». Ce n'est pas à proprement parler du logiciel, mais les moyens de le combattre sont voisins de ceux qui permettent de lutter contre les virus et autres malfaisances, parce que dans tous les cas il s'agit finalement d'analyser un flux de données en provenance du réseau pour rejeter des éléments indésirables.

6.7. Hoax

Il existe de faux virus, appelés hoaxes : un hoax se présente, en général, sous la forme d'un mail d'alerte contre un nouveau virus ; le message se réclame souvent d'un fabricant connu d'antivirus ou de matériel informatique, il signale un fichier dangereux et vous conseille de le détruire et demande qu'on diffuse largement l'information.

Le but des hoaxes est le simple plaisir, pour leurs concepteurs, de constater l'affolement et les encombrements provoqués par leur "plaisanterie". [3] [12]

7. MECANISMES DE PROTECTION

La sécurité informatique consiste aujourd'hui en grande partie en des méthodes défensives employées pour détecter et contrecarrer les possibles intrus et parer aux différents types de menaces possibles.

L'ajout dans le système de mécanismes ou fonctions supplémentaires qui forcent le respect des exigences de sécurité est l'une des approches utilisées dans la sécurité informatique. Nous introduisons, dans cette section, quelques uns des mécanismes de protection les plus utilisés.

7.1. L'authentification

Les mécanismes d'authentification sont employés pour s'assurer qu'un utilisateur particulier est bien celui qu'il prétend être. Deux types d'authentification sont généralement présents dans un système informatique : l'authentification de l'utilisateur lors de sa connexion au système et l'authentification de l'entité homologue lors de l'ouverture d'une communication entre deux entités du système.

Le procédé d'authentification le plus utilisé est le couple identifiant – mot de passe malgré ses faiblesses bien connues : un même utilisateur utilise souvent le même mot de passe pour accéder à diverses ressources, retenir et deviner, etc. [11]

7.2. Le contrôle d'accès

A supposer que grâce aux mécanismes d'authentification le système soit désormais capable de garantir qu'un utilisateur est bien celui qu'il prétend être, la prochaine étape est de fournir au système un moyen de vérifier qu'un utilisateur donné a effectivement le droit d'accéder aux objets : accéder aux ressources du système (par exemple les fichiers, les programmes, la mémoire, etc.) ou bien de communiquer avec un autre utilisateur.

Le mécanisme de contrôle d'accès détermine, pour chaque utilisateur, les différents modes d'accès (ou droits d'accès), comme la lecture, l'écriture, ou l'exécution, qu'il peut avoir sur chaque objet.

7.3. Le chiffrement de l'information (cryptographie)

La cryptographie est à la base de la plupart des mécanismes de sécurité. Elle a pour but de garantir la protection des communications transmises sur un canal public contre différents types d'adversaires. La protection des informations se définit en termes de confidentialité (les données transmises ne sont pas dévoilées à une tierce personne), d'intégrité (les données n'ont pas été modifiées entre l'émission et la réception) et d'authentification (l'**authentification de message** assure qu'il provient bien de la bonne entité et l'**authentification de personne**, aussi appelée **identification**, garantit qu'une entité qui cherche à s'identifier vis-à-vis d'un système informatique est bien celle qu'elle prétend être). [14]

7.3.1. Quelques définitions

- *Cryptographie* : étymologiquement « écriture secrète ». Aujourd'hui, c'est la science visant à protéger des messages à travers le chiffrement.
- *Chiffrement* : transformation à l'aide d'une clé d'un message en clair (texte clair) en un message incompréhensible (texte chiffré) pour celui qui ne dispose pas de la clé de déchiffrement;
- *Décrypter* : retrouver le message clair correspondant à un message chiffré sans posséder la clé de déchiffrement.

Les techniques cryptographiques sont essentiellement : le chiffrement symétrique et le chiffrement asymétrique.

7.3.2. Chiffrement symétrique

Dans le chiffrement symétrique, ou chiffrement à clé secrète, les algorithmes se fondent sur une même clé pour chiffrer et déchiffrer un message. Le problème de cette technique est que la clé, qui doit rester totalement confidentielle, doit être transmise au correspondant de façon sûre.

Les algorithmes de chiffrement symétrique les plus utilisés sont : DES (Data Encryption Standard), AES (Advanced Encryption Standard), IDEA (International Data Encryption Algorithm) et RC5 (Rivest Cipher).

7.3.3. Chiffrement asymétrique

Dans le chiffrement asymétrique, ou chiffrement à clé publique, la clé de cryptage et la clé de décryptage sont différentes. Chaque personne possède deux clés distinctes (une privée et une publique) avec impossibilité de déduire la clé privée à partir de la clé publique.

On peut utiliser des algorithmes à clé publique : en chiffrement/déchiffrement (cela fournit le secret), en échange de clés (ou des clés de session), et pour assurer l'authenticité d'un message ; l'empreinte du message est chiffrée à l'aide de la clé privée et est jointe au message, les destinataires déchiffrent ensuite le message chiffré à l'aide de la clé publique et retrouvent normalement l'empreinte. Cela leur assure que l'émetteur est bien l'auteur du message, on parle alors de la signature numérique.

Les algorithmes de cryptographie asymétrique les plus utilisés sont : RSA (Rivest Shamir Adleman) qui permet le chiffrement et la signature, DSA (Digital Signature Algorithm) qui permet la signature, et le Protocole d'échange de clés Diffie-Hellman. [15]

7.3.4. Fonction de hachage (Signature numérique)

Une fonction de hachage est une fonction qui permet à partir d'un texte de longueur quelconque, de calculer une chaîne de taille inférieure et fixe appelée condensé ou empreinte. Il est impossible de la déchiffrer pour revenir au texte d'origine, ce n'est donc pas une technique de chiffrement.

Les fonctions de hachage les plus utilisées sont : MD5 (Message Digest 5), SHA-1 (Secure Hash Algorithm), SHA-256.

L'empreinte d'un message ne dépasse généralement pas 256 bits (maximum 512 bits pour SHA-512) et permet de vérifier son intégrité. [16]

7.4. Anti-Virus

Les antivirus sont des logiciels conçus pour repérer les traces d'activité des virus, les bloquer et isoler ou supprimer les fichiers qui en sont responsables. Leur mode de fonctionnement est basé sur une veille permanente, à deux niveaux :

- Sur tout ordinateur, un programme antivirus doit être installé et actif.
- Cet antivirus doit être tenu à jour : la surveillance par l'antivirus se réfère à une base de données contenant les signes d'activité de tous les virus connus.

Chaque jour, de nouveaux virus apparaissent, inventés par des experts en programmation désireux d'éprouver leurs compétences ; en permanence, d'autres experts surveillent l'apparition de ces nouveaux programmes et conçoivent des antidotes.

7.5. Pare-feu (Firewall)

Un pare-feu est un système permettant de protéger un ordinateur ou un réseau d'ordinateurs des intrusions provenant d'un réseau tiers (notamment Internet). Il permet de filtrer les paquets de données échangés (en entrée et en sortie) avec le réseau.

7.6. Anti-spyware

Les anti-spywares sont comme les antivirus. Ils sont utilisés pour les spywares.

7.7. Autres mesures de protection

- Ne pas donner son adresse mail sur un site inconnu.
- Ne pas répondre aux messages de spam ni cliquer sur les liens qui prétendent désabonner de ces courriers.
- Lors de la réception d'un message douteux de type hoax, avant de supprimer un fichier essentiel du système et d'alerter tout le carnet d'adresses, se renseigner (on peut trouver, sur Internet, des sites d'information sur ces fausses alertes). [12]

8. CONCLUSION

Nous avons vu, dans ce chapitre, qu'un système informatique est exposé à des attaques qui peuvent perturber son fonctionnement, dont la plupart parvient de la connexion au réseau Internet. Nous avons vu aussi que l'attaque est l'exploitation d'une vulnérabilité dans le système informatique. Donc nous pouvons conclure que chaque système informatique est exposé à une famille d'attaques selon les vulnérabilités qu'il présente.

La plupart des attaques logiques que nous avons présentées sont communes pour presque tous les systèmes informatiques. Certains systèmes présentent d'autres vulnérabilités et donc ils sont exposés à d'autres types d'attaques. Parmi ces systèmes on trouve les systèmes embarqués.

Dans le chapitre suivant, nous allons présenter les systèmes embarqués et les différents aspects de sécurité dans ces systèmes.

Chapitre III :
La sécurité des systèmes
embarqués

1. INTRODUCTION

Grâce aux progrès de l'électronique et à la miniaturisation croissante, l'informatique ne se cantonne plus à l'ordinateur mais devient peu à peu omniprésente. Elle devient de plus en plus un domaine transparent dans le sens où elle disparaît tout en s'intégrant et se faufilant dans presque tous les domaines. On entend souvent parler d'informatique médicale, industrielle, du management, bioinformatique, etc. La forme à laquelle on fait illusion est la forme embarquée on parle alors de « systèmes embarqués ».

Ces systèmes, on les retrouve partout et non seulement dans les systèmes à grande échelle (comme ceux cités au préalable). On se réveille le matin grâce au radio réveil ; c'est un système embarqué, on utilise sa télécommande pour changer de programmes télé ; c'est un système embarqué, on parle à un ami sur un téléphone portable ; c'est un système embarqué, bref on ne saurait s'arrêter là dans la citation des systèmes embarqués qui nous entourent au quotidien, ils sont partout. Ces systèmes sont bourrés d'électronique complexe et d'informatique évoluée, et répondent exactement aux attentes pour lesquels ils ont été conçus et pas pour autre chose.

Grâce à notre dépendance croissante sur les systèmes embarqués, leur sécurité et fiabilité deviennent de plus en plus un problème. Ils sont utilisés pour saisir, stocker, manipuler, et accéder à des données sensibles (codes d'accès aux comptes, informations personnelles des carnets d'adresses, données médicales, etc.) dont la perte peut causer une sérieuse détresse ou de graves préjudices financiers.

La sécurité des systèmes embarqués est donc devenue une préoccupation importante. C'est un sujet en cours et peut rapidement devenir un problème encore plus grand que la sécurité des ordinateurs actuels. Ceci est dû aux contraintes liées aux dispositifs matériels (les ressources limitées et les contraintes de puissance) et une variété de vulnérabilités inhérentes rencontrés lors de la mise en œuvre des mesures de sécurité, ou bien au coût de la sécurité ; les fabricants tentent de réduire les coûts de production pour obtenir un avantage sur le marché pour les produits sensibles au prix.

Assurer la sécurité dans les systèmes embarqués se traduit en plusieurs défis de conception imposés par les caractéristiques uniques de ces systèmes. Ces caractéristiques font de l'intégration de mécanismes de sécurité classiques impraticable et nécessitent une meilleure compréhension du problème de sécurité tout entier. [17] [18]

Dans ce chapitre, nous allons présenter les systèmes embarqués, leur définition, leurs caractéristiques, leur importance et leurs domaines d'application, ainsi que leur architecture générale sous forme de couches tout en citons les composants matériels les plus communes de la plupart de ces systèmes. Et en fin nous allons aborder l'aspect sécurité dans les systèmes embarqués en décrivant les exigences de sécurité dans ces systèmes, les défis de conception d'un système embarqué sécurisé et les attaques possibles sur ces systèmes tout en concentrant sur les attaques logiques et la sécurité contre ces attaques.

2. LES SYSTEMES EMBARQUES

2.1. Définition

Selon la langue de Molière, le mot embarqué, est dérivé du verbe « embarquer » qui désigne le fait de « mettre quelque chose à bord d'un navire, un avion ou d'un véhicule ». Informatiquement parlant un système embarqué (appelé aussi un système enfui) peut être défini comme : « Un système électronique et informatique autonome, qui est dédié à une tâche bien précise ».

Il s'agit d'un composant électronique (une puce) équipé d'un logiciel, intégré dans un appareil ou un équipement qu'il contrôle, où il est difficile de distinguer entre le matériel et le logiciel. Il met en œuvre un processeur, à fin d'exécuter des tâches bien déterminées en réponse aux événements ou stimulus produits par l'environnement auquel il est intégré.

Le principal but de l'acquisition d'un système embarqué n'est pas l'utilisation de simples applications scientifiques ou commerciales traditionnelles, mais la réalisation d'opérations bien déterminées et prédéfinies pour lesquelles le système a été conçu (exemples : avionique, robotique, domotique, etc.). Car il ne s'agit nullement d'un simple PC grand public, possédant des entrées/sorties tel un clavier, une souris ou un écran standards. En général, les systèmes embarqués ne possèdent pas de ce type d'interfaces classiques, et leurs interfaces homme/machine peuvent se limiter à une led ou a un simple écran afficheur, comme ils peuvent être très sophistiqués et complexes comme une interface de vision de nuit. [17]

2.2. Domaines d'application

Les domaines dans lesquels on trouve des systèmes embarqués sont de plus en plus nombreux :

- Transport : automobile, aéronautique (avionique), etc.
- Astronautique : fusée, satellite artificiel, sonde spatiale, etc.
- Militaire : missile, etc.

- Télécommunication : set-top box, téléphonie, routeur, pare-feu, serveur de temps, téléphone portable, etc.
- Electroménager : télévision, four à micro-ondes, etc.
- Impression : imprimante multifonctions, photocopieur, etc.
- Informatique : disque dur, lecteur de disquette, etc.
- Multimédia : console de jeux vidéo, assistant personnel, etc.
- Guichet automatique bancaire (GAB).
- Equipement médical.
- Automate programmable industriel.
- Métrologie. [19]

2.3. Importance du marché de l'embarqué

L'omniprésence des systèmes embarqués dans notre vie est liée à la révolution numérique opérée dans les années 1970 avec l'avènement des processeurs. Les processeurs, de plus en plus rapides, puissants et bon marché ont permis cette révolution et aussi le boom du marché de l'embarqué. Ceci se confirme au travers de la loi empirique de Gordon Moore, cofondateur d'Intel, qui stipule que pour une surface de silicium donnée, on double le nombre de transistors intégrés tous les 18 mois !

La figure 3.1 montre cette évolution inexorable.

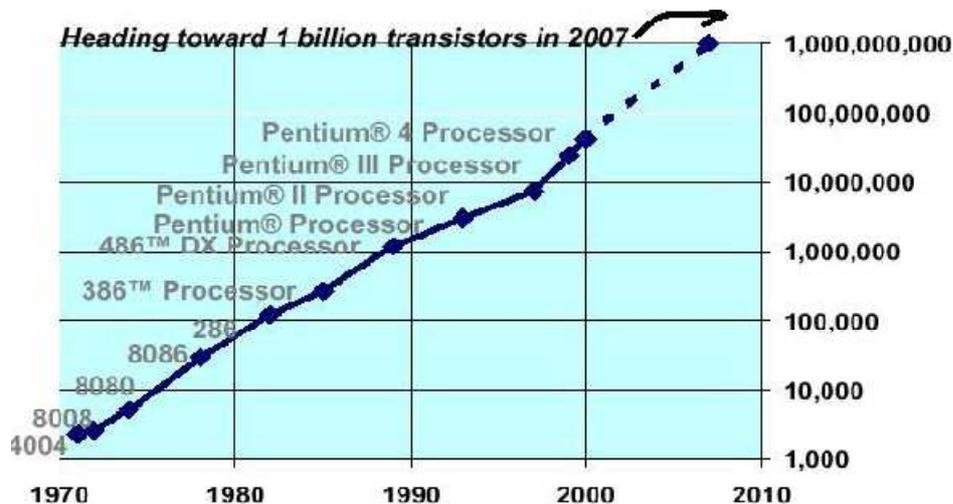


Figure 3.1. Loi empirique de Gordon MOOR pour le processeur Intel. [20]

En 1999, il a été vendu pour le marché de l'embarqué :

- 1,3 milliard de processeurs 4 bits.
- 1,4 milliard de processeurs 8 bits.

- 375 millions de processeurs 16 bits.
- 127 millions de processeurs 32 bits.
- 3,2 millions de processeurs 64 bits.

A côté de cela, à cette époque, il a été vendu seulement 108 millions de processeurs (famille x86) pour le marché du PC grand public !

Pour 2004, il a été vendu environ 260 millions de processeurs pour le marché du PC grand public à comparer aux 14 milliards de processeurs tout type confondu (microprocesseur, microcontrôleur, DSP (Digital Signal Processor)) pour le marché de l'embarqué.

On voit très bien donc, que la part de l'embarqué dans le marché des processeurs est considérable par rapport au PC, et l'on a réellement le droit de prendre conscience de l'importance du marché de l'embarqué. [20]

2.4. Les caractéristiques des systèmes embarqués

Un système embarqué doit pouvoir répondre à certaines caractéristiques. Ces caractéristiques peuvent être résumés en deux expressions « On doit pouvoir en dépendre » et « On doit pouvoir compter dessus », autrement dit un système embarqué doit avoir les caractéristiques suivantes, dont certains sont impératifs. [17]

2.4.1. Fiabilité

Les systèmes embarqués sont la plupart du temps dans des machines qui doivent fonctionner en continu pendant de nombreuses années, sans erreurs et, dans certains cas, réparer eux-mêmes les erreurs quand elles arrivent. C'est pourquoi les logiciels pour ces systèmes sont toujours développés et testés avec plus d'attention que ceux pour les PC. Les pièces mobiles non fiables (par exemple les lecteurs de disques, boutons ou commutateurs) sont proscrites. [19]

2.4.2. Faible coût

Vu que les systèmes embarqués sont partout et qu'ils sont utilisés par le grand public, il faudrait qu'ils aient des prix assez raisonnables pour être accessibles. [17]

2.4.3. Limités en ressources

Les contraintes de coût, de consommation énergétique et d'occupation physique limitent grandement le matériel embarquable. La puissance de calcul ainsi que la quantité de mémoire disponible sont ainsi très limités. [21]

2.4.4. Faible consommation

La consommation d'énergie joue un rôle important dans un système embarqué, notamment lorsqu'il s'agit de cibles mobiles utilisant des batteries pour leur fonctionnement. Il faut donc diminuer la consommation dans le système pour augmenter la durée de vie de la batterie et diminuer la dissipation thermique du système pour garantir la fiabilité.

2.4.5. Fonctionnement en temps-réel

Beaucoup de systèmes embarqués fonctionnent en temps-réel, ils sont toujours en interaction avec leur environnement, ils doivent prendre certaines décisions ou effectuer certains calculs dans un temps limite. Une faute temporelle peut entraîner une catastrophe dans le système. Il est donc nécessaire de vérifier l'exactitude temporelle du système (zéro défaut) avant sa mise en fonctionnement. [22]

2.4.6. Sûreté de fonctionnement

Les systèmes embarqués sont souvent critiques, s'il arrive que certains de ces systèmes subissent une défaillance, ils mettent des vies humaines en danger ou mettent en périls des investissements importants. Ils ne doivent donc jamais faillir. Par « *jamais faillir* », il faut comprendre toujours donner des résultats juste, pertinents et ce dans les délais attendus par les utilisateurs (machines et/ou humains) des dits résultats. [19]

2.4.7. Sécurité

Un système embarqué est en général en relation avec un environnement très variable et souvent hostile, pour des raisons physiques (la variation de la température, les vibrations et les chocs, la variation de l'alimentation, les radiations, etc.) ou humaines (malveillance). C'est pour cela que la sécurité (au sens de la résistance aux malveillances) et la fiabilité (au sens continuité de service) sont souvent rattachées à la problématique des systèmes embarqués.

Au cours du développement d'un système embarqué, le concepteur doit choisir entre plusieurs solutions d'architectures matérielles et logicielles répondant à des critères de performance et de sûreté de fonctionnement exprimés dans les spécifications. [22]

2.5. Architecture générale d'un système embarqué

Dans l'architecture d'un système normal, un ordinateur se compose de trois couches : la couche application, la couche système d'exploitation et la couche matérielle. De même, un système embarqué dispose de 3 couches. Chaque couche a la même fonctionnalité qu'un système normal, mais il y a des différences de sous composants du système. La figure 3.2 montre l'architecture en couches d'un système embarqué.

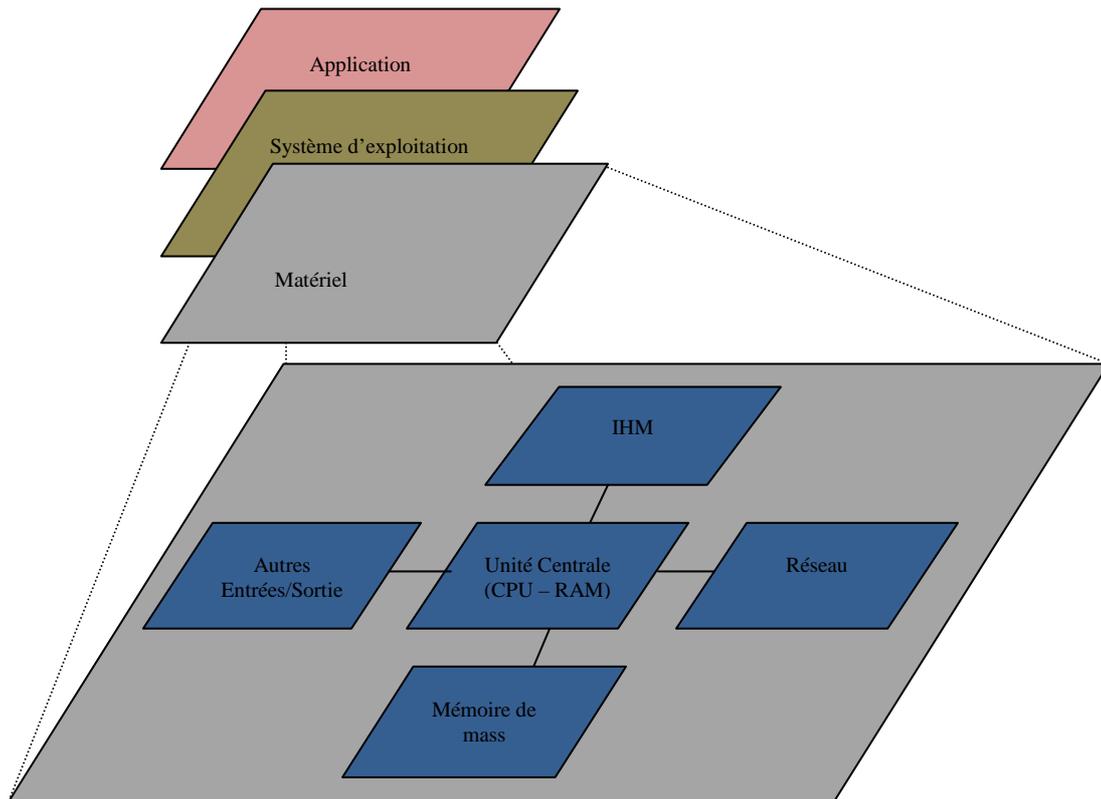


Figure 3.2. Architecture en couches d'un système embarqué. [23]

Pour les deux premières couches (application et système d'exploitation), il s'agit du logiciel. Le système d'exploitation est une couche logicielle sur laquelle va se placer l'ensemble des applications lancées par les utilisateurs. Il comprend des bibliothèques pour le développement, des drivers permettant aux applications d'accéder à des périphériques évolués et peut-être des interfaces pour contrôler les éléments.

Cette partie logicielle est stockée au niveau d'une mémoire, généralement une ROM. Elle est développée au début sur une machine hôte (avec un langage de programmation tel C/C++) le compilateur produit par la suite des fichiers *objets* qui vont être réunis pour former une bibliothèque. L'éditeur de lien reçoit en entrée ces fichiers et en fait une *image* exécutable, qui par la suite sera *mappée* sur la ROM du système embarqué.

Pour la dernière couche (matérielle), il s'agit d'un ensemble d'éléments physiques employés pour le traitement des données. On trouve des composants nécessaires qui s'appellent les composants permanents comme la (Read Only Memory) CPU et la mémoire vive RAM. On trouve aussi des composants supplémentaires qui varient selon le système embarqué et qui peuvent être : la carte dédiée aux applications spéciales, la mémoire de mass, le réseau, les entrées/sorties spéciales, etc. Ci-dessus nous allons citer quelques dispositifs : [17] [23]

- **CPU**

Les informations sont traitées au niveau d'une CPU, qui est le plus souvent un *microcontrôleur*, et non un *microprocesseur*. La différence réside dans le fait qu'un microcontrôleur est conçu de manière à ce qu'il ait le moins possible de ressources extérieures. Ce qui en autres termes, le rend plus rapide (pas de déperdition de signal), et à moindre coût, du fait que plusieurs éléments sont remplacés en un seul. [17]

- **RAM (Random Access Memory)**

Pour fournir un service, un système embarqué nécessite un minimum de mémoire de travail ou RAM. Cependant, le point important de cette mémoire est son coût élevé tendant à limiter son utilisation. [21]

- **Entrées**

- Les capteurs : ils sont appelés aussi senseurs et ils sont couplés avec des convertisseurs analogique/ numérique, ils captent le degré de luminosité, la vibration, la température à titre d'exemple. Ces capteurs sont responsables de l'acheminement des informations de l'environnement extérieur vers le système pour être traitées.
- Le clavier, les boutons poussoirs ou télécommandes (infrarouge, Bluetooth, radio, etc.).
- Les lecteurs de tags ou de codes barres.

- **Sorties**

- Les actionneurs : les actionneurs ou acteurs, couplés avec des convertisseurs analogique/ numérique, sont chargés par exemple d'allumer ou d'éteindre une led, de démarrer le contrôle de sécurité d'un moteur, de démarrer une pompe, etc.
- Les écrans et les afficheurs LCD (Liquid Crystal Display).
- Le système d'alarme ou la synthèse vocale.
- L'imprimante en tous genres comme papier, étiquettes, photos, etc.

- **Mémoire de mass**

- Le disque dur : microdrive à la taille environ 2,5 - 3,5 inches.
- La mémoire flash : FlashDisk, CompactDisk, DiskOnChip, Mémoire stick, clés USB.
- La mémoire ROM : CD, DVD, etc.
- Le disque à distance : NFS (Network File System), TFTP (Trivial File Transfer Protocol).

- IHM

- L'interface Homme/Machine d'un système embarqué varie selon sa fonction. Il y a ceux dont l'interface se limite à de simples affichages lumineux, et ceux dont l'interface est largement complexe comme dans le cas de cockpit d'avion. [17] [23]

3. LA SECURITE DANS LES SYSTEMES EMBARQUES

L'aspect sécurité est très important dans la conception des systèmes embarqués. Bien que les protocoles de sécurité et les algorithmes cryptographiques (vus dans le Chapitre II) adressent les considérations de sécurité d'une perspective fonctionnelle, plusieurs systèmes embarqués sont contraints par leurs environnements et par les ressources qu'ils possèdent. C'est la raison pour laquelle les considérations de sécurité étaient étendues d'une perspective fonctionnelle centrique à un problème de conception du système. [24]

3.1. Exigences de sécurité

La figure 3.3 cite les exigences communes de sécurité pour les systèmes embarqués, qui sont décrits comme suit :



Figure 3.3. Les exigences communes de sécurité des systèmes embarqués. [25]

- **Identification de l'utilisateur** : indique le processus d'authentifier les utilisateurs avant de leur permettre d'utiliser le système.
- **Accès sécurisé au réseau** : fournit une connexion au réseau ou un accès au service seulement si le dispositif est autorisé.

- **Communications sécurisées** : se composent d'authentification de communication entre les égaux, d'assurance de confidentialité et d'intégrité des données communiquées, d'empêchement de répudiation d'une transaction, et de protection de l'identité des entités communiqués.
- **Stockage sécurisé** : garantit la confidentialité et l'intégrité des informations sensibles stockées dans le système.
- **Contenu sécurisé** : impose les restrictions d'utilisation du contenu numérique stocké ou consulté par le système.
- **Disponibilité** : s'assure que le système peut exécuter sa fonction attendue et servir les utilisateurs légitimes à tout moment, sans être interrompu par des attaques de déni de service.

3.2. Défis de conception d'un système embarqué sécurisé

Les concepteurs d'un système embarqué doivent supporter de diverses solutions de sécurité afin de traiter une ou plusieurs des exigences de sécurité décrites ci-dessus. Ces exigences présentent des empêchements dans le processus de conception, qui sont brièvement décrits ci-dessous :

- **Brèches de traitement**

Dans les systèmes avec les ressources modestes de traitement et de mémoire, les demandes élevées de calculs pour les services de sécurité exigent normalement des changements dans la technologie (par exemple, couper le processus de code d'octet Java en deux phases) ou l'utilisation des composants dédiés.

- **Flexibilité**

Un système embarqué est souvent conçu pour répondre aux exigences d'exécution des multiples et divers protocoles de sécurité et standards pour supporter : des objectifs de sécurité, l'interopérabilité dans différents environnements, le traitement de sécurité dans différentes couches de réseau. En outre, avec des protocoles de sécurité constamment visés par des hackers, il n'est pas étonnant qu'ils continuent à se produire de nouvelles attaques. Il est souhaitable de permettre à l'architecture de sécurité d'être assez flexible (programmable) pour adapter facilement aux changements. Cependant, la flexibilité peut également causer des difficultés envers la sécurité.

- **Tamper-résistance**

Les attaques des logiciels malveillants tels que des virus et des chevaux de Troie sont les menaces les plus communes à n'importe quel système embarqué qui est capable d'exécuter des applications téléchargées. Puisque ces attaques peuvent compromettre la sécurité de tous points de vue (l'intégrité, l'accès aux données personnelles, la disponibilité), il est nécessaire de développer et déployer de diverses contre-mesures matériels/logiciels contre ces attaques.

- **Brèches d'assurance**

Il est bien connu qu'il est beaucoup plus difficile de construire des systèmes véritablement plus fiables que ceux qui fonctionnent simplement dans la plupart du temps. Les systèmes sûrs doivent pouvoir manipuler des situations qui peuvent se produire par hasard et ils relèvent un plus grand défi : ils doivent continuer à fonctionner sûrement en dépit des attaques réalisées par des adversaires intelligents qui cherchent intentionnellement des modes de défaillances indésirables. Lorsque les systèmes deviennent plus complexes, il y a inévitablement des modes de défaillance qui doivent être adressés.

- **Remarque**

Les deux facteurs, coût et batterie, influent également sur l'efficacité des mesures de sécurité d'un système embarqué.

3.3. Attaques sur les systèmes embarqués

Au niveau supérieur, les attaques sur les systèmes embarqués peuvent être classifiées dans deux larges catégories : attaques physiques et *side-channel*, et attaques logiques.

3.3.1. Attaques physiques et *side-channel*

Les attaques physiques et *side-channel* se réfèrent aux attaques qui exploitent l'implémentation physique du système et/ou exploitent les propriétés et défauts d'implémentation. Elles sont généralement classifiées en des attaques *invasives* et *non-invasives*.

Les attaques invasives ont pour but d'obtenir l'accès au dispositif pour observer, manipuler, et interférer dans les systèmes internes. Puisque les attaques invasives exigent typiquement une infrastructure relativement chère, elles sont assez difficiles à déployer. Un exemple d'une attaque invasive est les techniques de *micro-probing* (qui exigent généralement avoir accès aux composants au niveau de la puce afin d'intervenir et de manipuler le fonctionnement interne du système).

D'autre part, les attaques non-invasives n'exigent pas que le système doit être ouvert, et peuvent être réalisées sous plusieurs formes. Par exemple : les attaques d'induction de faute observent le comportement du système ciblé après avoir généré des erreurs ou des échecs sur lui par la manipulation de ses conditions d'exploitation (la tension d'alimentation, la température, les radiations, la lumière, etc.). D'autres attaques possibles comprennent les attaques Side-Channel comme les attaques de synchronisation et les attaques basés sur l'analyse de consommation qui s'appuient sur l'observation de la corrélation qui existe entre la consommation d'énergie par l'appareil, ou bien sur son profil de synchronisation, et les opérations exécutés ou les données secrètes manipulées (les clés cryptographiques).

3.3.2. Attaques logiques

Ces attaques sont multipliées par la capacité du système à télécharger et exécuter des codes d'applications. Elles sont essentiellement basées sur les logiciels malveillants (tels que les virus et les chevaux de Troie) et elles exploitent des faiblesses ou des bogues dans l'architecture globale (matériel/logiciel) du système aussi bien que des problèmes dans la conception de l'algorithme cryptographique ou du protocole de sécurité. Dans la pratique, les attaquants utilisent souvent une combinaison des plusieurs techniques pour atteindre leurs objectifs. Par exemple, pour un smartphone, du point de vue de l'attaquant, les objectifs visés sont : de rendre inutilisable le téléphone portable; de modifier le comportement du téléphone; d'accéder aux données sensibles; d'émettre des appels voix ou données à l'insu de l'utilisateur et d'outrepasser les politiques de gestion des droits numériques. [18] [25]

3.4. La sécurité contre les attaques logiques

Un aspect central et critique dans le problème de la sécurité des systèmes embarqués est la sécurité logique. Les attaques logicielles exploitent les vulnérabilités du système tels que les *portes dérobées* (en anglais *backdoor*, c'est une fonctionnalité inconnue de l'utilisateur légitime, qui donne un accès secret au logiciel), l'ignorance des standards usuels, la mauvaise programmation (ex. dépassement de tampon : en anglais, *buffer overflow*, c'est un bug par lequel un processus, lors de l'écriture dans un tampon, écrit à l'extérieur de l'espace alloué au tampon, écrasant ainsi des informations nécessaires au processus.), un serveur HTTP vulnérable car léger, etc.

Les mesures de sécurité fonctionnelles pour répondre aux exigences de sécurité se fondent typiquement sur les primitives cryptographiques mentionnées dans le Chapitre II, ou sur les mécanismes de sécurité qui emploient une combinaison de ces primitives d'une façon spécifique (les protocoles de sécurité, les certificats numériques, les protocoles de gestion des droits numériques, le stockage sécurisé et l'exécution sécurisée).

Ces mesures ne présentent pas une solution complète au problème de sécurité. Un problème de sécurité est plus susceptible de survenir en raison d'un problème dans une partie standard du système (par exemple, l'API sur le serveur) que dans certaines fonctions de sécurité données. C'est une raison importante pour laquelle la sécurité logicielle doit faire partie du cycle de vie complet du logiciel.

Les bonnes pratiques de la sécurité logique impliquent de penser à la sécurité plus tôt dans le cycle de vie du logiciel du système embarqué, de connaître et de comprendre les menaces communes sur les systèmes embarqués, de concevoir la sécurité, et de soumettre tous les artefacts logiciels à des analyses de risque objectives et des essais.

Notons que les risques peuvent surgir à tous les étapes du cycle de vie du logiciel, donc une analyse de risque, avec des activités périodiques de suivi et de surveillance de risque, sont fortement recommandées. [24]

4. CONCLUSION

Dans les présents systèmes embarqués, la sécurité n'est pas habituellement prise en compte au cours de la phase de conception du produit et elle est difficile à mettre en œuvre une fois que le produit est terminé. Même dans les cas où la sécurité a été une préoccupation depuis le début, le développeur doit faire face à des contraintes matérielles importantes lors d'inclusion des mesures de sécurité.

Il y a un nombre croissant de menaces à la sécurité sur les systèmes embarqués. Une garantie de sécurité à 100% n'existe plus, un attaquant a toujours assez de temps, des ressources et de motivation pour pouvoir pénétrer dans n'importe quel système. Pour cette raison, les fabricants doivent sécuriser leurs produits contre des menaces spécifiques et essayer d'atteindre un équilibre entre le coût de mise en œuvre de la sécurité et les bénéfices obtenus. [25]

Dans ce chapitre, nous avons présenté les attaques possibles sur un système embarqué, en général, et les mesures de sécurité contre ces attaques. Nous avons vu que les attaques utilisent les vulnérabilités présentes dans le système, on trouve des attaques communes pour tous les systèmes embarqués et des attaques qui sont spécifiques à un système particulier en fonction des vulnérabilités qu'il présente. Donc, les mesures de sécurité qu'on doit mettre en œuvre ne sont pas les mêmes pour tous les systèmes embarqués, ils se différencient d'un système à un autre selon les vulnérabilités présentes dans le système.

Dans le chapitre suivant, nous allons présenter un type particulier des systèmes embarqués « les Java Cards », et présenter les différents aspects de sécurité liés à ces systèmes.

Chapitre IV :

Java Card

1. INTRODUCTION

Depuis son apparition, la carte magnétique a montré ses énormes capacités. Mais, avec son utilisation répandue, la carte magnétique révèle de graves inconvénients jour après jour. Des données qui sont stockées dans cette carte peuvent facilement être lues et modifiées illégalement.

L'idée est d'utiliser la nouvelle technologie des circuits électroniques, pour faire cohabiter un moyen de stockage d'information et une puissance de calculs permettant une manipulation contrôlée et sécurisée de données. Développée vers la fin des années 70, la carte à puce est considérée comme une carte qui puisse surmonter ces inconvénients. Traditionnellement, la carte à puce a été employée seulement pour l'identification et le stockage électronique de données personnelles. Cependant aujourd'hui, elle a rapidement progressé pour devenir un support d'exécution de programmes qui contient un processeur totalement opérationnel. [26]

Au début de la carte à puce, les langages de programmation utilisés étaient l'assembleur et le langage C. Le problème avec ces langages est qu'ils sont dépendants de la carte sur laquelle va être implémenté le programme. Les développeurs de programmes ont rencontré de grandes difficultés et coûts pour exécuter leurs programmes sur plusieurs plates-formes. En mars 1996, un groupe d'ingénieurs de Schlumberger au Texas ont étudié la possibilité d'utiliser un langage de programmation de haut niveau pour les cartes à puce en préservant la sécurité des données. Leur première idée fût d'utiliser Java. Hélas, le système d'exécution ne pouvait pas être utilisé sur les cartes à cause de leur taille réduite. Pour résoudre ce problème, il était nécessaire d'identifier un sous-ensemble de Java. C'est ainsi qu'est né le langage Java Card, son environnement d'exécution, et sa machine virtuelle. [1]

La technologie Java Card a apporté des innovations majeures dans le monde des cartes à puce : les applications sont écrites en Java et sont portables sur toutes les Java Cards, les Java Cards peuvent exécuter des applications multiples, qui peuvent communiquer à travers des objets partagés, et en fin de nouvelles applications, appelées *applets*, peuvent être téléchargées sur la Java Card.

Aujourd'hui, une grande quantité de cartes à puces sont équipées avec une plate-forme Java Card et les émetteurs de cartes ont commencé à utiliser les possibilités offertes par cette technologie. Elles sont largement utilisées dans les cartes SIM (utilisées dans les téléphones mobiles), les cartes de guichets automatiques, et les services financiers (cartes de crédit, porte-monnaie électroniques, etc.).

La possibilité de charger du code exécutable après la délivrance de la carte soulève des questions de sécurité majeures. Il n'y a à priori aucune raison d'estimer que le nouveau code chargé a été développé de manière à ne pas interférer avec les applications existantes. Dès lors, un des principaux problèmes pour le déploiement de nouvelles applications est de pouvoir fournir l'assurance que ces nouvelles applications ne menacent pas la sécurité de la carte en tant que plate-forme d'exécution. On se doit donc d'assurer que leur exécution ne risque pas de compromettre l'intégrité de la carte ou la confidentialité des autres données qui y sont stockées. [5]

Dans ce chapitre nous allons présenter les cartes à puces, tout en concentrant sur les cartes intelligentes (Smart Card) et leur architecture. En suite nous allons présenter les Java Cards, qui sont des Smart Cards utilisant un sous-ensemble de la technologie Java, nous allons décrire comment les aspects de Java sont utilisés dans les Java Cards (le langage, l'environnement d'exécution, la machine virtuelle) et comment se déroule l'installation d'applications externes sur ces cartes. Enfin, nous allons présenter l'aspect sécurité dans les Java Cards, les problèmes apportés par le chargement d'applications externes et les éléments de sécurité disponibles dans Java Card pour traiter ces problèmes.

2. LES CARTES A PUCES

Une carte à puce est une carte en matière plastique, voire en papier ou en carton, de quelques centimètres de côté et moins d'un millimètre d'épaisseur, portant au moins un circuit intégré (une puce) capable de contenir de l'information. Le circuit intégré peut contenir un microprocesseur capable de traiter cette information, ou être limité à des circuits de mémoire non volatile et, éventuellement, un composant de sécurité. Les cartes à puce sont principalement utilisées comme moyens d'identification personnelle (carte d'identité, badge d'accès aux bâtiments, carte d'assurance maladie, carte SIM), de paiement (carte bancaire, porte-monnaie électronique) ou de preuve d'abonnement à des services prépayés (carte de téléphone, titre de transport). [27]

2.1. Types de base

Il existe différents types de cartes à puces. On distingue les cartes à mémoire des cartes à microprocesseur.

Les cartes à mémoire ne possèdent pas de CPU intégré et ne peuvent par conséquent pas traiter de l'information. Elles servent simplement d'unité de stockage (comme les cartes téléphoniques). Les données déjà présentes sont traitées par des fonctions simples préprogrammées sur la puce. Comme le circuit a un nombre limité de fonctions qui ne peuvent être reprogrammées, les cartes à mémoire ne peuvent pas être réutilisées et sont donc jetées à la fin de leur utilisation.

A l'inverse les cartes à microprocesseur, qui sont appelées cartes intelligentes (*smart cards* en anglais), contiennent une CPU. Elles permettent de réaliser des fonctionnalités bien plus grandes et un niveau de sécurité plus élevé. L'information n'est ainsi pas directement disponible aux applications externes. Le processeur n'autorise l'accès aux données que sous certaines conditions (mot de passe, cryptage,...). Elles sont utilisées dans tous les domaines qui nécessitent un haut niveau de sécurité (banques, téléphones portables, etc.). Les applications sont multiples et très flexibles et ne sont limitées que par l'espace mémoire du circuit.

Dans les cartes à microprocesseur on peut également faire la distinction entre les cartes à contact et les cartes sans contact. Les cartes à contact doivent être insérées dans un appareil et dans un sens particulier (ex : cartes de crédit). Les cartes sans contact, elles, n'ont pas besoin d'être placées dans une machine. Elles communiquent avec l'extérieur grâce à une antenne incorporée dans la carte. L'énergie peut être fournie par une batterie interne ou captée par l'antenne. Elles transmettent alors leurs données à un appareil par ondes électromagnétiques. Ces cartes sont donc très populaires dans des situations nécessitant des transactions rapides. Toutefois la brièveté de la transaction ne permet de transmettre qu'une quantité limitée de données. [28]

Dans la suite, nous utilisons le terme Smart Cards pour les cartes à microprocesseur avec contact qui sont les plus courantes.

2.2. Architecture des Smart Cards

2.2.1. Composants physiques

Une Smart Card est constituée de trois parties :

- Un support en plastique, aux dimensions désormais connues sous le nom de « format carte de crédit », supportant les contraintes d'une utilisation quotidienne ;
- Un module de contact, assurant la liaison entre la puce et son lecteur ;
- La puce elle-même, située sous le module de contact. [4]

- **Normes**

Dans le domaine des cartes à puce, la demande de normalisation est très importante. En effet, un émetteur peut être réticent à investir dans une technologie potentiellement incompatible avec les générations futures. C'est pour cela que de nombreuses normes existent dans les différents domaines d'applications, la plus connue de ces normes est la norme ISO 7816.

La norme ISO 7816 est la norme de référence pour les cartes à puce. Elle spécifie entre autres les éléments physiques et dimensionnels des supports plastiques, la signification et la localisation des contacts, les protocoles et contenus des messages de bas et haut niveaux échangés avec les cartes. Par exemple la norme ISO 7816-1 définit les caractéristiques physiques de la carte plastique : *dimensions 85 x 54 mm* et *épaisseur : 0,76 mm*. [1]

La puce rassemble en un bloc monolithique les composantes courantes d'un ordinateur. On y trouve un microprocesseur et divers types de mémoires.

2.2.1.1.CPU

Le microprocesseur présent sur les Smart Cards est le plus souvent un microcontrôleur 8 bits (fondé sur des registres, des bus d'adresses, ou des bus de données de taille de 8 bits) utilisant le jeu d'instructions Motorola 6805, ou Intel 8051 avec une vitesse d'horloge à 5 MHz. Certaines cartes permettent de multiplier la vitesse d'horloge par 2, 4 ou 8 atteignant une vitesse d'horloge de 40 Mhz. Des microcontrôleurs à 16 et 32 bits commencent à apparaître sur les nouvelles Smart Cards.

2.2.1.2.Mémoires

Les Smart Cards contiennent habituellement trois sortes de mémoires : ROM, EEPROM et RAM.

- La ROM (Read-Only Memory) est utilisée pour stocker les programmes fixes de la carte, les routines du microprocesseur et les données permanentes. On ne peut ainsi ni les modifier, ni les supprimer. Aucune alimentation n'est nécessaire pour conserver les données. Sa taille est le plus souvent de 32 ko, mais on peut trouver des cartes contenant 300 ko ou plus (sur les cartes modernes haut de gamme).
- L'EEPROM (Electrical Erasable Proramnable Read-Only Memory), comme la ROM peut conserver les données quand la mémoire n'est plus alimentée. La différence est que l'on peut modifier les données stockées dans la mémoire. En fait c'est l'équivalent pour les Smart Card du disque dur d'un PC. Sa taille est le plus souvent de 16 ko.

- La RAM (Random Access Memory) est utilisée comme espace temporaire pour modifier et stocker les données. Son contenu n'est pas préservé à l'arrêt de l'alimentation. Sa capacité est seulement de 512 octets à 2 ko.

D'autres technologies de mémoire sont également en train d'apparaître sur les Smart Cards comme la mémoire Flash. Celle-ci est en effet plus efficace en espace et en temps que la mémoire EEPROM. [28]

2.2.1.3. Points de contact

Une Smart Card possède en général huit points de contact :

- Le point Vcc (volt en courant continu) d'alimentation. La tension varie en général de 3 à 5 volts.
- Le point RST (ReSeT) utilisé pour réinitialiser le microprocesseur. Cela permet une "réinitialisation Chaude" et évite d'avoir à sortir et réintroduire la carte dans l'appareil en cas d'erreur (réinitialisation froide).
- Le processeur ne possédant pas d'horloge interne, celle-ci est située sur le point CLK (CLOCK).
- Le point GND (GROUND) est la masse. Sa tension est zéro.
- Le point Vpp (Peak-Point Voltage) est optionnel et n'est plus utilisé sur les cartes actuelles. Il permettrait de contrôler la tension d'entrée.
- Le point I/O (Input/Output) est le point d'entrée-sortie permettant de transmettre les données à l'extérieur en mode semi-duplex : un seul sens de communication est autorisé à un moment donné.
- Enfin les deux points RFU (Reserved for Future Use) ne sont là qu'en réserve pour une utilisation future.

2.2.2. Dispositif d'acceptation de cartes « Card Acceptance Device (CAD) »

Une Smart Card ne possède pas sa propre alimentation, son afficheur, ou son clavier. Elle interagit avec un dispositif d'acceptation de carte (CAD) à l'aide d'une interface de communication, fournis par la collection de points de contact.

Le CAD (appelé aussi lecteur de carte, lecteur de dispositif, ou terminal de carte) peut être un ATM (Automated Teller Machine), un EFTPOS (Electronic Funds Transfer at Point of Sale) ou n'importe quel PC connecté à un lecteur de carte. Il sert à conduire l'information vers et depuis la carte. Fondamentalement, toute application de la carte est divisée en deux parties, l'une s'exécute sur la carte et l'autre sur le terminal.

Le terminal communique avec la carte en utilisant un mécanisme de paquets appelé Application Protocol Data Unit (APDU). Les cartes à puce sont des communicateurs réactifs, c.à.d. qu'ils n'initialisent jamais une communication, ils répondent seulement aux APDUs du CAD. Le modèle de communication est un model commande-réponse, c.à.d. la carte reçoit une commande APDU, effectue le traitement demandé par la commande, et retourne une réponse APDU. [29][30]

2.2.3. Système d'exploitation

Le système d'exploitation est le logiciel de commande d'une Smart Card, qui interprète et exécute les différents ordres élémentaires que cette carte puisse réaliser. Il est généralement écrit en C ou en langage d'assemblage et il est intégré dans la ROM en cours de la fabrication de la puce. Le système d'exploitation réalise les fonctions suivantes :

- Gestion des échanges entre la carte et le monde extérieur, notamment le protocole d'échanges.
- Gestion des différents fichiers et des données à l'intérieur de la mémoire.
- Contrôle des accès aux informations et aux fonctions (par exemple sélection de fichier, lecture, écriture, modification de données).
- Gestion de la sécurité de la carte et de la mise en œuvre des algorithmes cryptographiques.
- Assurance de la fiabilité du fonctionnement, notamment cohérence et intégrité de certaines données, ruptures de séquences et reprise des erreurs.
- Gestion du cycle de vie de la carte dans ses différentes phases (fabrication, personnalisation, utilisation, fin de vie). [26]

2.2.4. Applications

Au début des Smart Cards, les applications ont été développées avec des langages de bas niveau et elles sont intégrées dans la carte au moment de la fabrication. Les langages de bas niveau sont spécifiques à la puce et les programmes qui sont écrits pour une carte ne peuvent pas être chargés sur une autre carte.

Comme les fabricants de puces ont commencé à développer leurs propres outils logiciels indépendamment des normes ISO 7816, et il est devenu difficile pour les programmeurs que les applications soient dépendantes de la plate-forme. Ceci est devenu un grand obstacle qui ralentissait les progrès de la programmation des applications pour les Smart Cards. [30]

Depuis quelques années, les systèmes d'exploitation ouverts ont fait leur apparition. Ces systèmes apportent la possibilité à la carte d'héberger plusieurs applications et d'en charger de nouvelles après sa mise en service. Ces systèmes d'exploitation sont évolués et adaptés aux impératifs de sécurité que requiert cette nouvelle fonctionnalité. Parmi ceux-ci, citons Smart Card for Windows de Microsoft, MultOS du Consortium Maosco, Camille de l'Université de Lille, Open Platform de Visa, .NET Card de Axalto (ancienne division Smart Cards de Schlumberger) et Java Card de Sun Microsystems qui est le plus utilisé.

Ces systèmes ne contiennent que le SE, avec une couche de chargement d'applications: les applications sont développées dans un langage intermédiaire (JavaCard (dérivé de Java), Multos (dérivé de C), WfSC (dérivé de Visual Basic)) puis compilées dans un format de chargement. [4]

3. JAVA CARD

La technologie Java Card est l'adaptation de Java pour les cartes à puce et d'autres dispositifs à mémoire limitée. C'est l'une des dernières plates-formes pour carte à puce, et la plus populaire. Conçue par Schlumberger, Java Card a été reprise en 1996 par Sun Microsystems afin de compléter son offre de développement basée sur la technologie Java au domaine de la carte à puce.

La technologie Java Card définit une plate-forme sécurisée pour cartes à puce, portable et multi-application qui incorpore beaucoup d'avantages du langage Java. Cette plate-forme est articulée autour d'un sous-ensemble du langage Java, d'une machine virtuelle appelée Java Card Virtual Machine (JCVM), d'un environnement d'exécution appelé Java Card Runtime Environment (JCRE), et d'un ensemble de bibliothèques accessibles par des Application Programmer Interface (APIs).

On appelle une carte à puce Java Card, toute carte à puce possédant une plate-forme basée sur la technologie Java Card. Elle est dite « ouverte » car elle permet de charger et d'exécuter des programmes écrits en Java. Contrairement aux cartes à puce traditionnelles, les programmes exécutés par la carte ne sont pas forcément fournis par l'émetteur de carte. [4]
[26]

3.1. Le langage Java Card

Dû à la faible capacité de mémoire des cartes à puces, le langage utilisé sur les plateformes Java Cards ne peut être qu'une partie du langage Java. Il peut se résumer à un sous-ensemble de Java auquel on aurait retiré les fonctionnalités trop gourmandes en calcul ou en espace mémoire pour être exploitées sur une carte à puce. Il s'agit d'un langage à objet, fortement typé, sans pointeurs explicites, à exceptions, à héritage de classe simple et à héritage d'interface multiple. Les éléments supportés et non supportés du langage Java par Java Card sont présentés dans le tableau 4.1.

Eléments supportés	Eléments non supportés
<ul style="list-style-type: none"> – Types primitifs : boolean, byte, short. – Tableau à une dimension. – Packages Java, classes, interfaces, exceptions. – Programmation orientée objet : héritage, méthodes virtuelles, surchargement, création d'objets dynamiques, règles de liaisons. – Le mot clé int et le type integer sur 32 bits sont optionnels. 	<ul style="list-style-type: none"> – Types primitifs : long, double, float. – Caractères et chaînes. – Tableaux multidimensionnels. – Chargement de classes dynamiques. – Garbage collection et finalisation. – Threads. – Sérialisation d'objets. – Clonage d'objets.

Tableau 4.1. Eléments supportés et non supportés du langage Java. [28]

La similitude avec le langage Java est si grande que le code source Java Card est écrit dans la syntaxe Java et compilé par l'intermédiaire d'un compilateur Java standard (le code source doit cependant se contenter de n'utiliser que les fonctionnalités supportées par Java Card). Le programme compilé porte la dénomination d'*applet* et offre une portabilité comparable aux programmes Java de par l'utilisation d'un ensemble de code octet commun (bytecode).

Les programmes Java Card ne sont pas représentés comme les programmes Java par un fichier class (.class). Le fichier class produit par la compilation du code source subit une transformation (dans le sens de la simplification) afin d'adapter le format de programmes aux ressources minimalistes disponibles sur la carte à puce. Le programme au format Converted APplet (CAP) qui résulte de cette phase de transformation est le programme qui va être chargé sur la carte, il est directement exécutable, a déjà subi l'édition de lien et contient simplement les informations nécessaires à son exécution. La chaîne de compilation est donnée sur la figure 4.1.

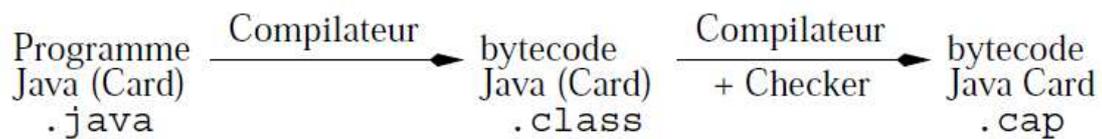


Figure 4.1. Compilation d'un programme Java Card. [1]

- ***Fichier CAP et Fichier d'Exportation***

La technologie Java Card introduit deux nouveaux formats binaires de fichiers qui permettent l'exécution d'applications Java Card indépendamment de la plate-forme :

Le fichier CAP (Converted Applet) qui contient une représentation binaire exécutable de toutes les classes d'un package Java. Le format CAP est le format dans lequel les logiciels sont chargés sur la Java Card. Les fichiers CAP permettent par exemple le chargement dynamique des classes d'applet après la conception de la carte.

Les fichiers d'exportation ne sont pas chargés sur les Smart Cards et ne sont donc pas directement utilisés par l'interpréteur. Ils sont produits et consommés par le convertisseur à des fins de vérification. On peut considérer les fichiers d'exportation comme les fichiers .h du langage C. Ils ne contiennent en effet que des informations sur les APIs publiques des classes et des signatures de méthodes et de champs. Ils peuvent donc être laissés en accès libre aux utilisateurs de l'applet sans pour autant en révéler son code source. [4] [28]

- **Remarque**

Le convertisseur et l'interpréteur sont deux composantes de la machine virtuelle Java Card qu'ils vont être représentés dans la section suivante.

3.2. Architecture d'une Java Card

Le système d'architecture de Java Card est illustré dans la figure 4.2.

La description des différents composants de l'architecture d'une Java Card est donnée dans le tableau 4.2.

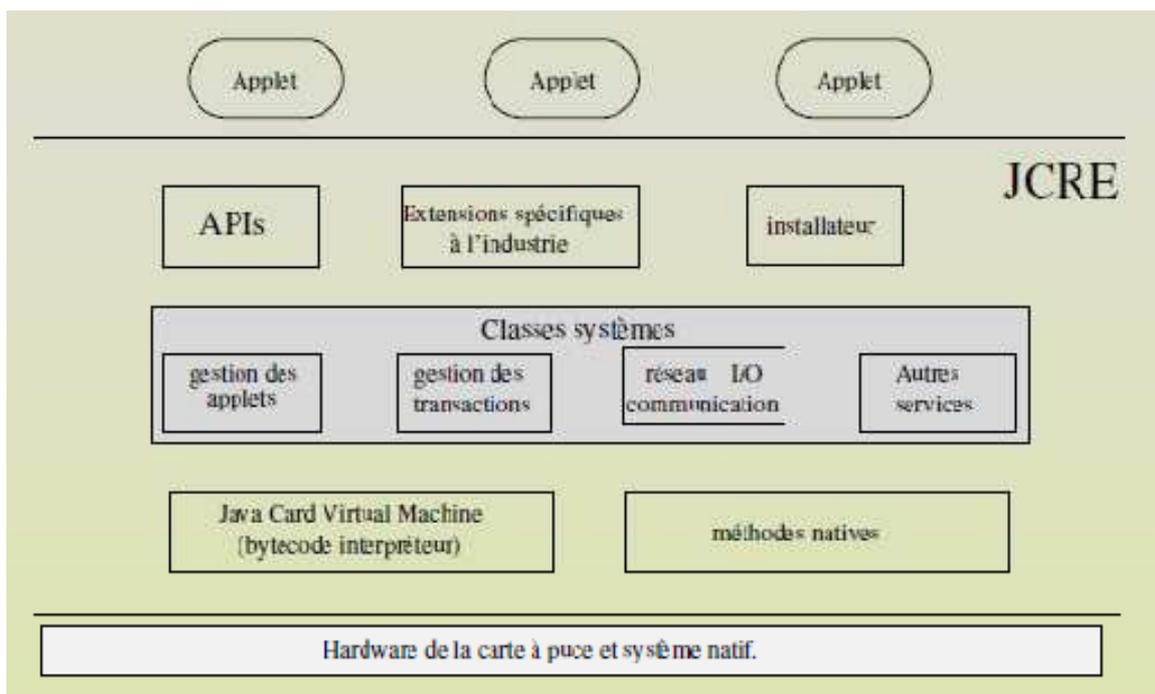


Figure 4.2. Architecture d'une Java Card. [31]

Composant	Description
Hardware de la carte à puce et système natif	Traitent les opérations de bas niveau comme le transfert des données physiques, la gestion des fichiers, la communication et l'exécution d'instructions.
Environnement d'exécution Java Card (JCRE)	Gère les ressources de la carte, la communication avec le réseau, l'exécution des applets et leur sécurité. Il consiste en : la machine virtuelle Java Card (l'interpréteur de bytecode), l'installateur, le framework qui contient l'API Java Card, les extensions industrielles spécifiques, les méthodes natives et les classes système du JCRE.
Machine virtuelle Java Card (JCVM)	Fournit l'exécution de bytecode et le support du langage Java Card, contenant la manipulation d'exception.
Installateur	Permet le téléchargement sécurisé des applets sur la carte après qu'elle est faite et délivrée au titulaire de carte. L'installateur coopère avec le programme d'installation hors-carte. Ensemble, ils accomplissent la tâche de chargement du contenu binaire des fichiers CAP.
Framework	L'ensemble des classes qui implémentent l'API. Il contient des packages de noyaux et d'extensions. Responsable du dispatching des APDUs, de la sélection d'applet, du contrôle d'atomicité, et d'installation des applets.

Application Programming Interface (APIs)	Définit la convention d'appels par lesquels une applet accède aux méthodes natives et au JCRE.
Classes système	Agissent en tant que l'exécutif du JCRE. Elles sont en charge de la gestion des transactions, la gestion de la communication entre les applications hôtes et les applets Java Card, et le contrôle de la création, la sélection et désélection d'applet. Pour exécuter les tâches, les classes système typiquement appellent les méthodes natives.
Méthodes natives	Offre les services de carte : entré/sortie, cryptographie, et allocation de mémoire.
Extensions industrielles spécifiques	Des classes ajoutées qui étend des applets installés dans la carte.
Applets	Des programmes écrits en langage Java Card pour être utilisés sur la carte à puce.

Tableau 4.2. Description des composants d'architecture d'une Java Card. [26]

3.2.1. La machine virtuelle Java Card

La machine virtuelle Java Card (Java Card Virtual Machine ou JCVM) est basée sur les notions de piles pour la manipulation des valeurs et de tas pour le stockage des objets. Elle comporte un contexte d'exécution par méthode appelée.

Tout comme la machine virtuelle Java, elle exécute du code sous forme de codes octet (bytecode), c'est-à-dire une représentation par des mnémoniques d'instructions élémentaires, avec éventuellement des opérandes. Ces codes octet seront ensuite traduits dans le langage du microprocesseur de la carte à puce par le système d'exploitation, qui associe à chaque code octet un comportement calculatoire correspondant à sa sémantique. [4]

Pour des raisons dues aux ressources limitées de la carte, la JCVM est subdivisée en deux parties séparées : l'une sur la carte, l'autre sur l'ordinateur destiné à la lire. La partie sur la carte contient l'interpréteur de bytecode alors que le convertisseur est situé sur l'ordinateur. Pris ensemble ils implémentent toutes les fonctions de la machine virtuelle Java nécessaires au sous-langage Java des Java Cards.

3.2.1.1. *Le convertisseur*

Contrairement à la JVM qui ne traite qu'une classe à la fois, l'unité de conversion du convertisseur est un package. Les fichiers .class sont produits par le compilateur Java à partir du code source. Ensuite, le convertisseur prétraite tous les fichiers .class contenus dans un package et les convertit en un fichier CAP (Converted Applet). Le fichier CAP est ensuite chargé sur une Java Card et exécuté par l'interpréteur.

Durant le processus de conversion, le convertisseur exécute des tâches qui sont normalement exécutées par la JVM au moment du chargement des classes :

- Il vérifie que les images des classes Java sont bien formées.
- Il vérifie qu'il n'y a pas de violations du sous-langage Java.
- Il initialise les variables statiques.
- Il convertit des références symboliques de classes, des méthodes et des champs en une forme plus compacte supportée par les Java Cards.
- Il alloue, stocke et crée des structures de données de la machine virtuelle pour représenter les classes.

Le convertisseur ne prend pas seulement en entrée les fichiers .class à convertir en .CAP mais aussi un ou plusieurs fichiers d'exportation. Si le package importe des classes d'autres packages le convertisseur charge aussi les fichiers d'exportation de ces packages. Le convertisseur produit donc un fichier CAP et un fichier d'exportation du package converti.

3.2.1.2. *L'interpréteur*

L'interpréteur, en fournissant un support d'exécution du langage Java, permet aux applets d'être indépendantes du matériel (hardware). Il exécute en particulier les tâches suivantes :

- Exécution des instructions du bytecode et des applets.
- Contrôle de l'allocation mémoire et de la création d'objets.
- Sécurise le temps d'exécution.

La machine virtuelle Java Card a été décrite comme comprenant le convertisseur et l'interpréteur. Elle est toutefois définie informellement comme étant la partie située sur la carte, à savoir l'interpréteur actuellement décrit. Ainsi concernant les Java Cards, on peut faire l'analogie entre l'interpréteur et la machine virtuelle, contrairement à la machine virtuelle Java qui elle, inclue le convertisseur et l'interpréteur. [28]

3.2.2. Installeur Java Card et programme d'installation hors-carte

L'interpréteur Java Card ne charge pas lui même les fichiers CAP, il exécute uniquement le code trouvé dans ces fichiers. Dans la technologie Java Card, les mécanismes de téléchargement et d'installation d'un fichier CAP sont incorporés dans une unité appelée *l'installeur*.

L'installeur Java Card réside dans la carte et coopère avec un programme d'installation hors-carte. Le programme d'installation hors-carte transmet le binaire exécutable dans un fichier CAP à l'installeur situé sur la carte via un dispositif d'acceptation des cartes (CAD). L'installeur écrit le binaire en mémoire de la carte à puce, il le lie avec les autres classes qui ont déjà été placés sur la carte, et crée et initialise toutes les structures de données qui sont utilisées en interne par l'environnement d'exécution Java Card. L'installeur et le programme d'installation et comment ils se rapportent au reste de la plate-forme Java Card sont illustrés dans la figure 4.3.

La répartition des fonctions entre l'interpréteur et l'installeur des fichiers CAP maintient l'interpréteur petit et fournit la flexibilité pour les implémentations d'installeur. [31]

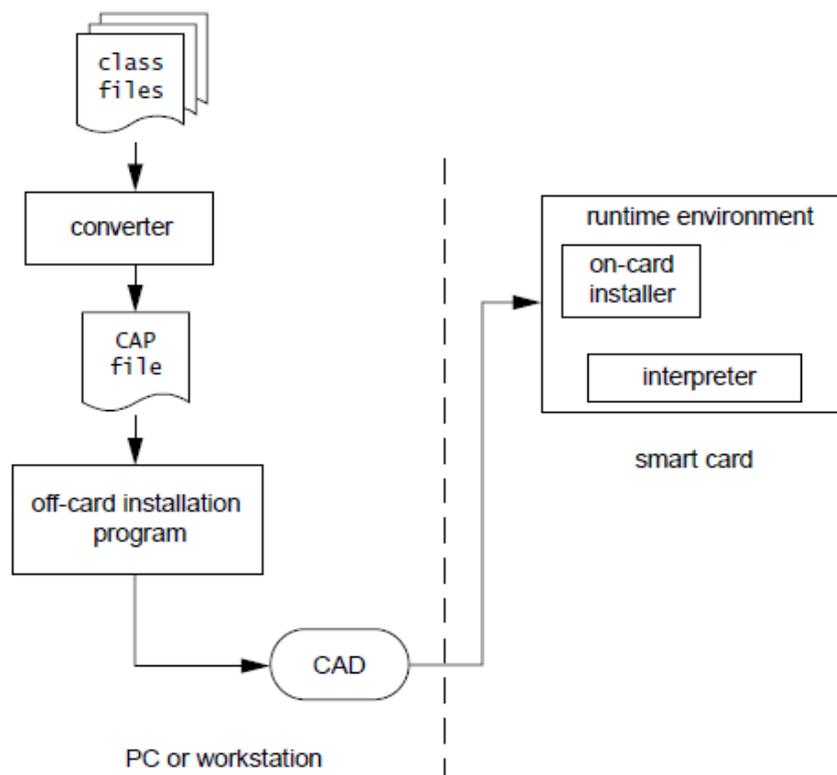


Figure 4.3. Installeur Java Card et programme d'installation hors-carte. [31]

3.2.3. L'environnement d'exécution Java Card

L'environnement d'exécution Java Card (Java Runtime Environment ou JCRE) gère les ressources de la carte, la communication avec le réseau, l'exécution des applets et leur sécurité. Le JCRE est une sorte de système d'exploitation universel pour la carte à puce. Comme on peut le voir dans la figure 4.2, le JCRE se situe au dessus du hardware de la carte à puce et du système natif. Il consiste en : la machine virtuelle Java Card (l'interpréteur de bytecode), l'installateur, le framework qui contient l'API Java Card, les extensions industrielles spécifiques, les méthodes natives et les classes système du JRCE. Le JCRE sépare habilement les applets de la propriété de la technologie des Smart Cards des vendeurs. Les applets sont ainsi plus faciles à écrire et sont indépendantes de l'architecture privé des Smart Cards. [28]

3.2.3.1. Cycle de vie d'un JCRE

Dans un PC ou un poste de travail, la machine virtuelle Java s'exécute comme un processus du système d'exploitation. Les données et les objets sont créés dans la RAM. Lorsque le processus d'exploitation se termine, les applications Java et leurs objets sont automatiquement détruits.

Dans une Java Card, la JCVM s'exécute au sein du JCRE. Le JCRE est initialisé au moment de l'initialisation de la carte. L'initialisation du JCRE n'est effectuée qu'une seule fois pendant la durée de vie de la carte. Durant ce processus, le JCRE initialise la machine virtuelle et crée des objets pour fournir les services du JCRE et la gestion des applets. Quand les applets sont installées, le JCRE crée des instances d'applets, et les applets créent des objets pour le stockage de données.

La plupart des informations sur une Smart Card doivent être préservées, même lorsque l'alimentation est retiré de la carte. La technologie de mémoire persistante (comme les EEPROMs) est utilisée pour réaliser cette préservation. Les données et les objets sont créés dans la mémoire persistante. La durée de vie du JCRE est équivalente à la durée de vie complète de la carte. Lorsque l'alimentation est coupée, la machine virtuelle est seulement suspendue. L'état du JCRE et des objets créés sur la carte sont préservés.

La prochaine fois que la carte est sous tension, le JCRE redémarre l'exécution de la machine virtuelle par le chargement des données de la mémoire persistante. Une notion subtile ici est que le JCRE ne reprend pas le fonctionnement de la machine virtuelle à l'endroit exact où la tension a été perdue. La machine virtuelle est réinitialisée et exécutée depuis le début de la boucle principale. La réinitialisation du JCRE diffère de son initialisation, car il préserve des applets et des objets créés sur la carte. Pendant la réinitialisation, si une transaction n'a pas été préalablement terminée, le JCRE effectue tout nettoyage nécessaire pour amener le JCRE dans un état cohérent. [31]

3.2.3.2. Comment fonctionne le JCRE durant une transaction ?

La période pendant laquelle la carte est insérée dans une machine est appelée session CAD. Durant une session CAD, le JCRE opère comme une Smart Card en établissant une communication APDU (Application Protocol Data Unit) d'entrée-sortie avec un serveur. Les APDUs sont des paquets de données échangés entre des applets et une application hôte. Chaque APDU contient soit un ordre du serveur à l'applet, soit une requête de l'applet au serveur.

Après l'initialisation du JCRE, le JCRE entre dans une boucle, en attente d'un ordre APDU du serveur. Celui-ci envoie des ordres APDUs à la Java Card par l'intermédiaire d'une communication utilisant les points de contacts d'entrée-sortie.

Quand une commande arrive, le JCRE sélectionne soit un applet à exécuter comme indiqué dans la commande, soit envoie directement la commande à un applet déjà sélectionné. L'applet sélectionné prend alors le contrôle et exécute la commande APDU. Une fois la commande exécutée, l'applet envoie une réponse à l'application hôte et redonne le contrôle au JCRE. Le procédé se répète à l'identique quand une nouvelle commande arrive. [28]

3.2.4. Les APIs Java Card

Les APIs Java Card consistent en un ensemble de classes spécialisées dans la programmation d'applications de Smart Card conformément à la norme ISO 7816.

Les APIs contiennent trois packages principaux et un package d'extension. Les trois packages principaux sont les packages `java.lang`, `javacard.framework`, et `javacard.security`. Le package d'extension est `javacardx.crypto`.

Les classes dans ces APIs sont compactes et succinctes. Elles incluent des classes adaptées à la plate-forme Java Card fournissant un support sur le langage et des services de cryptographie. Elles contiennent également des classes spécialement conçues pour supporter le standard ISO 7816 des Smart Cards.

3.2.4.1. Le package *java.lang*

Ce package est un sous-ensemble du package Java équivalent *java.lang*. Les classes supportées sont *Object*, *Throwable*, et quelques classes d'exceptions reliées à la machine virtuelle. Beaucoup de méthodes Java ne sont toutefois pas disponibles dans ces classes. La classe Java Card *Object* par exemple ne définit qu'un constructeur par défaut et la méthode *equals*.

3.2.4.2. Le package *javacard.framework*

Le package *javacard.framework* est essentiel. Il fournit l'architecture des classes et des interfaces pour le fonctionnement basique des applets Java Card. Plus important, il définit la classe de base *Applet* qui donne le cadre d'exécution et d'interaction des applets durant leur fonctionnement. Son rôle est similaire à celui de la classe *Applet* d'un browser internet.

Deux autres classes importantes de ce package sont la classe *APDU* qui sert aux protocoles de transmission et la classe *PIN* qui gère les mots de passe.

3.2.4.3. Le package *javacard.security*

Ce package est utilisé pour les fonctions de cryptographie de la plate-forme Java Card. Son implémentation est basée sur le package Java du même nom *java.security*. Il définit notamment la classe *keyBuilder* et des interfaces variées représentant le système cryptographique de clés (DES) et (DSA et RSA).

3.2.4.4. Le package *javacardx.crypto*

Comme dit précédemment ce package est un package d'extension. Il contient des classes de cryptographie et des interfaces dont les spécifications sont régulées par les Etats-Unis. Il définit la classe abstraite *Cipher* pour l'encryption et la décryption des données. [26]

3.3. Les Applets Java Card

L'entité exécutable en Java Card est l'applet. Les applets Java Card ne doivent pas être confondues avec les applets Java juste parce qu'elles sont tous nommées applets. Une applet Java Card est un programme Java qui adhère à un ensemble de conventions qui lui permettent d'être exécuté dans l'environnement d'exécution Java Card. Une applet Java Card n'est pas destinée à fonctionner dans un navigateur. La raison pour laquelle le nom applet a été choisi pour les applications Java Card est qu'elles peuvent être chargées dans le JCRE après que la carte a été fabriquée. C'est, à la différence des applications dans de nombreux systèmes embarqués, les applets n'ont pas besoin d'être brûlées dans la ROM en cours de la fabrication. Plutôt, elles peuvent être téléchargées dynamiquement sur la carte à une date ultérieure.

3.3.1. Identificateur d'application (AID)

Dans la technologie Java Card, à chaque instance d'une applet est assigné un AID. De même chaque paquet Java est identifié par un AID. Ceci est la convention de nommage définie par la norme ISO 7816.

Un AID est une séquence d'octets utilisée pour l'identification unique d'applets et de paquets. Cette séquence d'octet est composée de deux pièces distinctes. Voici comment elles sont organisées :

RID (National Registered Application Provider)	PIX (Proprietary Application Identifier Extension)
5 octets	0 – 11 octets

Tableau 4.3. Identificateur d'application (AID). [32]

Le RID est fournit aux compagnies par l'ISO. Chaque compagnie a donc un RID unique. Ensuite ce sont ces compagnies qui complètent l'AID avec le PIX de leur choix. A l'arrivée, chaque applet (ou paquet) a donc un AID unique : [RID, PIX].

L'AID d'un paquet et l'AID par défaut d'une applet sont spécifiés dans le fichier CAP. Ils sont fournis au convertisseur lorsque le fichier CAP est généré.

L'applet peut s'enregistrer auprès du JCRE en utilisant l'AID par défaut trouvé dans le fichier CAP, ou elle peut choisir un autre. Les paramètres d'installation peuvent être utilisés pour fournir un AID alternatif. [31][32]

3.3.2. Processus de développement d'une Applet

La figure 4.4 illustre le processus de développement d'une applet.

Le développement d'une applet Java Card commence comme avec tout autre programme Java: un développeur écrit une ou plusieurs classes Java et compile le code source avec un compilateur Java, produisant un ou plusieurs fichiers de classe.

Ensuite, l'applet est exécutée, testée et déboguée dans un environnement de simulation. Le simulateur simule l'environnement d'exécution Java Card sur un PC ou un poste de travail. Dans l'environnement de simulation, l'applet s'exécute sur une machine virtuelle Java, et donc c'est les fichiers de classe de l'applet qui sont exécutés. Le simulateur peut utiliser de nombreux outils de développement Java (la machine virtuelle, un débogueur, et d'autres outils) et permet au développeur de tester le comportement de l'applet et de voir rapidement les résultats de l'applet sans passer par le processus de conversion. Ainsi on ne peut pas tester le comportement de l'applet vis-à-vis du firewall, des objets temporaires et persistants et d'autres caractéristiques (on présentera le firewall et les objets temporaires et persistants plus tard dans ce chapitre).

L'étape suivante est la conversion des paquets Java afin d'obtenir les fichiers CAP.

La dernière étape permet de tester complètement les applets Java Card sur un émulateur. Un émulateur Java Card émule le JCRE sur une station de travail et comporte une JCVM. Ainsi le comportement de l'applet sera le même sur l'émulateur que sur la carte après son chargement.

La plupart des simulateurs et des émulateurs vont de paire avec un débogueur permettant ainsi au développeur de mettre des points d'arrêts, de suivre les changements d'états, etc. Lorsque l'applet est finalisée, l'étape suivante sera son installation sur la carte. [26] [32]

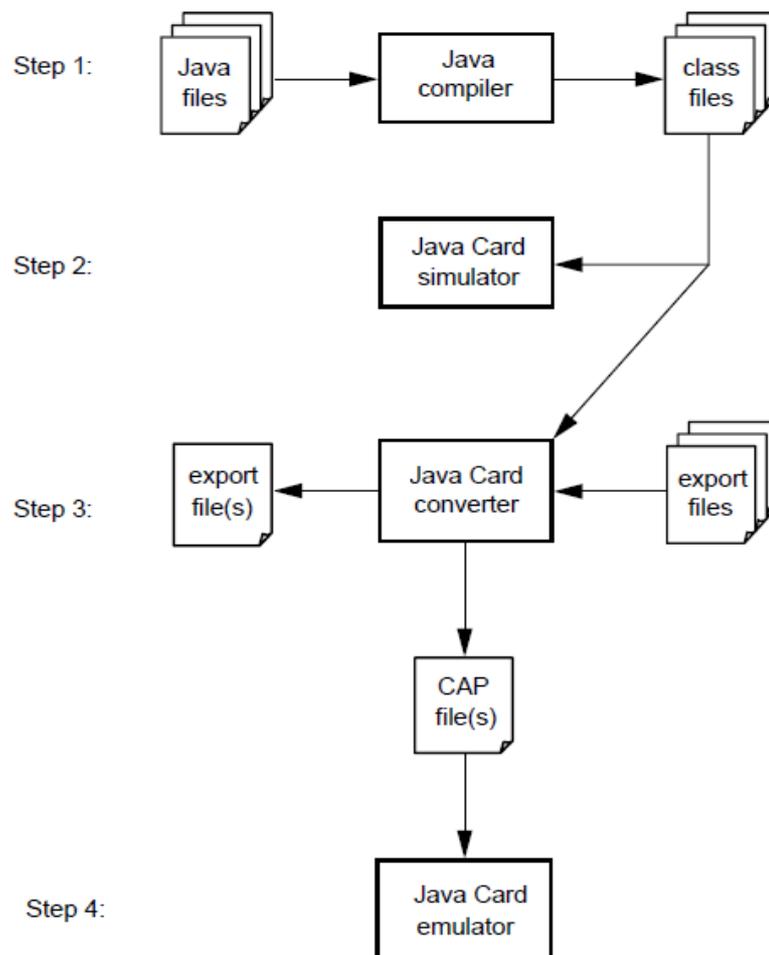


Figure 4.4. Processus de développement d'une Applet. [31]

3.3.3. Installation et exécution d'une applet

Quand une Java Card est fabriquée, le système natif et le JCRE sont brûlés dans la ROM. Ce processus d'écriture des composantes permanentes dans la mémoire non mutable d'une puce est appelé *masquage*.

Les classes d'applet Java Card peuvent être masquées dans la ROM avec le JCRE et d'autres composants du système pendant le processus de fabrication des cartes. Les instances d'applets sont instanciés dans l'EEPROM lors de l'initialisation du JCRE ou à un stade ultérieur. Ces applets sont appelées les applets de ROM.

Alternativement, les classes d'applet Java Card peuvent être téléchargées et écrites dans la mémoire mutable (EEPROM) après que la carte a été fabriquée. Ces applets sont appelées les applets *Post-Issuance* (*après-emission*).

Dans le reste de cette section nous allons nous concentrer sur l'installation des applets post-issuance.

L'installation d'une applet se réfère au processus de chargement de ses classes contenues dans un fichier CAP sur la Java Card, leur combinaison avec l'état d'exécution du JCRE, et la création d'une instance de l'applet pour la mener dans un état de sélection et d'exécution.

Sur une plate-forme Java Card, l'unité de chargement et installable est un fichier CAP. Un fichier CAP se compose d'un ensemble de classes qui composent un paquet Java. Une applet minimale est un package Java avec une seule classe dérivée de la classe *javacard.framework.Applet*. Une applet plus complexe constituée de plusieurs classes peut être organisée dans un seul package ou dans un ensemble de packages Java.

Pour charger une applet, le programme d'installation hors-carte prend le fichier CAP et le transforme en une séquence de commandes APDUs, qui transportent le contenu du fichier CAP. En échangeant les commandes APDUs avec le programme d'installation hors-carte, l'installateur situé sur la carte écrit le contenu du fichier CAP en mémoire persistante de la carte et lie les classes contenues dans ce fichier avec d'autres classes qui se trouvent sur la carte. L'installateur crée et initialise toutes les données qui sont utilisées en interne par le JCRE pour supporter l'applet. Si l'applet nécessite plusieurs packages pour être exécutée, chaque fichier CAP est chargé sur la carte.

Dans la dernière étape lors de l'installation d'applet, l'installateur crée une instance d'applet et enregistre cette instance auprès du JCRE. Pour faire ceci, l'installateur appelle la méthode d'installation de l'applet :

```
public static void install(byte[] bArray, short offset, byte length)
```

La méthode *install* est une méthode qui doit être mise en œuvre par l'applet, c'est un point d'entrée à applet, similaire à la méthode principale *main* dans une application Java. Cette méthode appelle le constructeur de l'applet pour créer et initialiser une instance de l'applet. Le paramètre *bArray* de la méthode *install* fournit les paramètres d'installation pour l'initialisation d'applet. Ces paramètres sont envoyés à la carte avec le fichier CAP, et le format et le contenu de ces paramètres sont définis par le développeur d'applet.

La méthode *install* peut être appelée plusieurs fois pour créer des instances multiples de l'applet. Chaque instance de l'applet est identifiée par un AID unique. Dans l'environnement Java Card, une applet peut être écrite et exécutée sans même savoir comment ses classes sont chargées. La seule responsabilité d'une applet lors de l'installation est d'implémenter la méthode *install*.

Après l'initialisation et l'enregistrement de l'applet auprès du JCRE, elle peut être sélectionnée et exécutée. La sélection d'une applet a lieu quand le JCRE reçoit une APDU de sélection dans laquelle l'identifiant correspond à l'AID de l'applet. Il appelle la méthode *select* de l'applet demandée. Si la méthode *select* retourne *true* alors elle devient l'applet active et elle peut envoyer l'APDU de sélection à la méthode *process* de l'applet.

Toutes les APDUs reçues par le JCRE sont envoyées à la méthode *process* (excepté les APDU de sélection). En fonction de l'APDU, la méthode *process* exécute l'opération demandée et le cas échéant renvoie une réponse.

Quand une APDU de sélection est reçue par le JCRE, il appelle la méthode *deselect* de l'applet active avant de sélectionner l'autre applet. Cela permet à l'applet d'effectuer des opérations avant de passer dans un état inactif.

Chaque applet s'exécute dans un espace qui lui est propre. La JCVM doit assurer cette propriété. Il existe des mécanismes de partage, qui permettent à une applet d'accéder à des services offerts par une autre applet ou par le JCRE. Cette caractéristique est assurée par le « firewall ». [1][31]

3.3.4. Structure d'une Applet

Le code d'une applet respecte la structure d'un code Java classique. La classe principale doit s'étendre de la classe *javacard.framework.Applet* et elle doit implémenter les méthodes publiques suivantes : *install*, *select*, *process*, *deselect*. [32]

Le code suivant est obtenu de [33], il présente un code simple qui ne réalise effectivement aucune opération, mais qui contient les éléments principaux que l'on retrouve dans une applet :

```
package demo_applet ;
// packages importés
// importation spécifique pour l'API Javacard
import javacard . framework .*;
public class example_applet extends javacard . framework
Applet {
    /**
    Constructeur par défaut
    @param baBuffer paramètres d'installation
    @param sOffset offset de départ
    @param bLength longueur en octets
    */
    protected example_applet ( byte [] baBuffer , short sOffset ,
    byte bLength ) {
    register ( baBuffer , ( short ) ( sOffset + 1), ( byte )
    baBuffer [ sOffset ]);
    }
    /**
    Methode pour installer l'applet.
    @param baBuffer paramètres d'installation
    @param sOffset offset de départ
    @param bLength longueur en octets
    */
    public static void install ( byte [] baBuffer , short sOffset
    , byte bLength ) {
    // création d'une instance de l'applet
    new example_applet ( baBuffer , sOffset ,bLength ). register
    ();
    }
    /**
    @le booléen retourné doit toujours être true .
    */
    public boolean select () {
    /** @todo : ACTION DE SELECTION A EXECUTER */
    // retourne l'état de la désélection
```

```
    return true ;
}
// Invoquée par le système dans le processus de désélection
public void deselect () {
    /** @todo : ACTION DE DESELECTION A EXECUTER*/
}
/**
Méthode qui traite une APDU entrante
@voir APDU
@param apdu APDU entrante
@exception ISOException avec les octets de réponse défini dans
l'ISO 7816-4
*/
public void process ( APDU apdu ) {
    /** @todo : METTRE L'ACTION A EXECUTER ICI */
} }
```

4. LA SECURITE DANS JAVA CARD

Une carte à puce peut être vue comme un support sécurisé de manipulation de données sensibles, car elle stocke des données d'une façon sûre et elle les manipule d'une manière fiable pendant la durée des transactions.

En général, le but d'un attaquant sur une carte à puce est de pouvoir accéder aux informations et aux secrets contenus dans la carte ou tout simplement de nuire aux applications embarquées afin de tester le niveau de sécurité de la carte.

Une Java Card se caractérise par la possibilité de charger du code exécutable après sa délivrance. Cette possibilité soulève des questions de sécurité majeures. Il n'y a à priori aucune raison d'estimer que le nouveau code chargé a été développé de manière à ne pas interférer avec les applications existantes. Dès lors, un des principaux problèmes pour le déploiement de nouvelles applications est de pouvoir fournir l'assurance que ces nouvelles applications ne menacent pas la sécurité de la carte en tant que plate-forme d'exécution. On se doit donc d'assurer que leur exécution ne risque pas de compromettre l'intégrité de la carte ou la confidentialité des autres données qui y sont stockées. [5]

Dans la suite de ce chapitre, nous allons présenter les risques de sécurité dans les cartes à puce en général (y compris les Java Card) en expliquant les motivations des attaquants et leurs cibles d'attaques, ensuite nous allons présenter les attaques qui peuvent être menées sur les Java Cards et qui sont de deux types physiques et logiques, nous nous concentrons principalement sur les attaques logiques provenant de la possibilité de charger du code exécutable dans une Java Card après sa délivrance, et nous présentons les mécanismes de sécurité disponibles sur les Java Cards contre ces attaques. Et en fin, nous montrerons les façons les plus utilisées pour obtenir un code mal typé (qui est un type d'attaque) sur une Java Card.

4.1. Les Risques

De manière générale, la mise en place de mécanismes de sécurité doit être précédée par une évaluation des risques de sécurité. En particulier, il est essentiel de savoir *contre qui* et *contre quoi* il faut se protéger.

Dans cette section, nous nous intéressons aux risques de sécurité des cartes à puce. Nous commençons par introduire les motivations guidant les attaquants, puis nous présentons les cibles visées.

4.1.1. Motivations

Les risques de sécurité peuvent être classés en fonction des motivations guidant l'attaquant. De manière générale, les motivations d'un attaquant d'une carte à puce sont du même style que celles guidant un attaquant d'un système informatique quelconque. Schématiquement, nous pouvons classer ces motivations en cinq catégories :

- Le gain financier,
- L'obtention de données,
- L'usurpation d'identité,
- Le vandalisme,
- La notoriété.

Le gain financier peut être direct ou indirect. Par exemple, dans le cas d'un porte-monnaie électronique, l'attaquant va chercher à créditer ou à bloquer les opérations de débits sur son porte-monnaie (gain financier direct). En revanche, dans le cas des cartes téléphoniques, des cartes de télévision à péage ou des cartes d'accès à un service en ligne de type WWW, il s'agit, pour l'attaquant, d'obtenir l'accès gratuit à un service (gain financier indirect).

L'objectif visé par l'obtention de données est l'atteinte à la vie privée ou l'accès à des informations sensibles de type commercial ou militaire. Ainsi, dans le cas d'une carte santé, les informations médicales contenues dans la carte peuvent être utilisées à mauvais escient par l'attaquant (ex. divulgation des informations).

De manière générale, l'usurpation d'identité permet d'accéder frauduleusement à un emplacement physique ou à un système. Au delà du simple accès, l'attaquant va également pouvoir effectuer toutes les opérations que l'utilisateur légitime est autorisé à faire.

L'objectif du vandalisme est de détruire des éléments du système ou d'aboutir à un déni de service. Accessoirement, le résultat attendu est souvent une contrepublicité pour le système ou le service.

Finalement, la motivation guidant les attaquants peut simplement être la reconnaissance et/ou la médiatisation.

4.1.2. Cible d'attaques

Nous nous concentrons maintenant sur les différentes cibles d'une attaque visant les cartes à puce. Nous identifions les attaques suivantes :

- La lecture ou la modification du code exécutable ;
- La lecture ou la modification des données ;
- La lecture ou la modification des secrets ;
- La modification du comportement de la carte (par un autre moyen que la modification du code exécutable).

Notons que la distinction entre données et secrets est arbitraire, mais permet de souligner la différence entre les informations *applicatives* (qui sont accessibles hors de la carte sous certaines conditions) et les clés et codes secrets qui ne doivent jamais être accessibles en dehors de la carte.

La lecture du code permet d'obtenir certains algorithmes secrets. La modification de code est une cible fréquente qui, si elle est atteinte permet de prendre le contrôle de la carte, de ses données, de ses secrets et de son comportement.

Le but de la lecture de données ou de secrets est évident. Celui de la modification d'information aussi.

La modification des secrets est moins courante, mais peut servir dans certaines attaques de type DFA (Differential Fault Analysis).

4.2. Attaques sur les Java Cards

Les attaques sur une carte à puce sont de deux types : physiques et logiques.

4.2.1. Attaques physiques

Les attaques physiques sont des attaques menées sur la partie électronique de la carte. Elles reposent sur l'observation ou la modification du support d'exécution des programmes, c'est-à-dire sur le microprocesseur et ses connexions externes (masse, alimentation, horloge, port de communication série et remise à zéro).

Une première classe d'attaques physiques sont les attaques invasives. Ces attaques supposent l'accès aux composants élémentaires et aux bus internes du microprocesseur afin de lire/écrire/effacer des données. Il en résulte que la carte à attaquer est généralement détruite et inutilisable.

Les attaques invasives supposent l'utilisation d'équipements rares et coûteux, tel qu'un FIB (Focus Ion Beam), et habituellement destinées à faire de la mise au point ou du test de composants chez les fondeurs. Précisons que l'utilisation de tels équipements requière un niveau de connaissance et de compétence/pratique élevé. De ce fait, le risque lié aux attaques invasives est souvent considéré comme peu élevé.

La seconde classe d'attaques physiques sont les attaques non invasives. En opposition avec les attaques invasives, les attaques non invasives ne nécessitent pas de détruire la carte à puce cible. Ainsi, certaines attaques non invasives peuvent être réalisées au détriment du porteur de la carte. De ce fait, les risques liés aux attaques non invasives sont généralement considérés comme réels. Parmi les attaques non invasives, nous distinguons les attaques basées sur l'observation de l'environnement de la carte de celles basées sur sa perturbation.

L'observation consiste essentiellement à détecter et exploiter des canaux cachés pouvant laisser passer de l'information sur l'état interne de la carte, et en particulier, pour acquérir de l'information sur les secrets contenus dans la carte. Actuellement, ce sont essentiellement les canaux cachés temporels ou l'analyse du signal de tension d'alimentation qui sont exploités (attaques de type SPA (Simple Power Attack) ou DPA (Differential Power Attack)). Il est également fort probable que le rayonnement électromagnétique provoqué par le fonctionnement du microprocesseur constitue une source d'information.

La perturbation des entrées/sorties offre de nombreuses possibilités d'attaques. En appliquant des valeurs hors normes de façon continue ou transitoire, il est possible de perturber le microprocesseur et de modifier sélectivement une partie de la RAM, des registres ou bien de provoquer un décodage erroné d'une instruction. La plupart du temps, ce genre d'expérience n'aboutit à rien d'exploitable, mais, en systématisant les essais, il est possible de découvrir des comportements hautement intéressants. Il peut s'agir, par exemple, de la transmission de données après la limite du *buffer* d'entrées/sorties (donc potentiellement de secrets), ou de la modification en RAM des droits d'accès permettant ensuite d'accéder à des informations protégées.

Enfin, les attaques de type DFA consistent à induire des fautes lors de la manipulation de secrets (par exemple, lors d'une signature RSA) et à comparer les différents résultats produits. Elles peuvent permettre de remonter, très rapidement, aux secrets manipulés. Leur mise en pratique reste à démontrer et pourrait s'appuyer sur des techniques du paragraphe précédent ou sur des irradiations (photons ou autres particules).

Les attaques DFA, comme les précédentes, peuvent être effectuées dans un cadre invasif ou combinées avec des attaques invasives, ce qui accroît alors leurs capacités. [34]

4.2.2. Attaques logiques

Les attaques logiques visent le logiciel embarqué dans les cartes à puce. Il s'agit souvent d'exploiter des erreurs de conception ou de codage qui permettent à l'attaquant de contourner certains mécanismes de protection, ou de détourner l'utilisation de certaines fonctionnalités de la carte à puce.

Signalons que les attaques logiques peuvent éventuellement être combinées avec des techniques décrites dans la section précédente.

Dans une carte qui ne permet pas le chargement d'application après sa délivrance, l'attaquant est soit le développeur du logiciel embarqué (ex. l'émetteur de la carte), soit une entité extérieure qui va essayer d'exploiter les failles de sécurité existant dans le logiciel embarqué.

Dans le premier cas, l'attaquant a une parfaite connaissance du code, et a même eu le loisir d'intégrer volontairement une chausse-trappe (trapdoor) dans le logiciel embarqué. Bien que ce risque soit réel, nous pouvons raisonnablement considérer ce type d'attaque comme marginale. En effet, la découverte d'une telle chausse-trappe aurait pour conséquence de définitivement décrédibiliser le développeur, et ainsi, l'empêcher de nuire par la suite.

Dans le second cas, l'attaquant n'a pas nécessairement une bonne connaissance du code, mais a pu, par utilisation ou *reverse-engineering* déduire des informations sur le comportement de la carte. Il va alors essayer d'exploiter les informations acquises afin de trouver une faille se traduisant soit par la possibilité de contourner les mécanismes de protection de la carte, soit par le détournement de l'utilisation de fonctionnalités de la carte.

A titre d'exemple, considérons une carte à microprocesseur utilisée comme porte-monnaie électronique. L'objectif d'un attaquant est soit d'inhiber l'opération de débit, soit de pouvoir créditer illégalement son porte-monnaie. L'opération de débit peut être inhibée si, par exemple, la carte renvoie un acquittement avant d'effectuer le débit au sein de la carte : en arrachant la carte au moment opportun, le débit sera validé en externe, mais ne sera pas effectué en vérité.

Un attaquant peut être à même de créditer son porte-monnaie électronique si, par exemple, l'opération de crédit peut être, d'une manière ou d'une autre, rejouée. C'est le cas lorsque les informations contenues dans le message de crédit ne sont pas suffisantes pour garantir la fraîcheur (*freshness property*) du message (pas de date, compteur pas assez large, etc.).

Une Java Card se caractérise par la possibilité de charger du code exécutable après sa délivrance. En plus des attaques qui peuvent être menées sur une carte qui ne permet pas le chargement d'applications, la Java Card est sujet à d'autres types d'attaques provenant de cette fonctionnalité.

Plus précisément, quatre types de failles de sécurité peuvent être identifiées pour ces cartes :

- Les failles de sécurité externes aux cartes permettant à l'attaquant de charger du code malintentionné et/ou d'interagir sur du logiciel embarqué.
- Les failles de sécurité au niveau de la plate-forme : celles-ci recouvrent les problèmes de cloisonnement des applications, de la correction de la plate-forme (ce qui peut être vu comme la sécurité, au sens classique, du système d'exploitation), ainsi que de la qualité des services de sécurité rendus par la plate-forme (chiffrement, résistance aux attaques physiques).
- Les failles de sécurité applicatives : celles-ci exploitent généralement des failles de sécurité au niveau de la plate-forme, et dépendent principalement de la bonne conception et du bon codage de l'application.
- Les failles de sécurité liées au partage des objets entre les différentes applications.

Ainsi, outre la possibilité d'attaquer chaque application (vue comme du logiciel embarqué), y compris le système d'exploitation, l'attaquant a la possibilité d'exploiter des failles de sécurité externes à la carte, et d'interagir avec plusieurs applications simultanément, et ainsi contourner des mécanismes de sécurité, ou détourner l'utilisation de fonctionnalités.

En outre, contrairement aux cartes qui ne permettent pas le chargement d'applications, le risque qu'un fournisseur d'application intègre volontairement une chausse-trappe ou un Cheval de Troie ne peut être négligé dans le cas des Java Cards. En effet, une fois le logiciel malicieux au sein d'une carte, il pourra réaliser une attaque sans être détecté, ou sans être suspecté. [5] [34]

4.3. Éléments de sécurité disponibles dans Java Card

Les mécanismes de sécurité contre les attaques logiques dans la plate-forme Java Card sont basés sur ceux dans Java (présentés dans l'annex), tout en s'adaptant à Java Card, et en lui ajoutant des protections supplémentaires, spécifiques à la plate-forme Java Card. Il existe plusieurs éléments de sécurité disponibles et offertes par la plate-forme Java Card.

4.3.1. Sécurité du langage

Le langage Java Card étant un sous ensemble du langage Java, il convient de faire un bref rappel des bases de la sécurité du langage Java :

- Le langage Java est fortement typé. Ainsi on ne peut pas convertir des entiers en pointeurs.
- Les variables doivent être initialisées avant d'être utilisées.
- Le niveau d'accès de toutes les classes, méthodes et champs est strictement contrôlé par les modificateurs d'accès. Par exemple, une méthode peut être déclarée comme étant publique, protégée ou privée. Une méthode privée ne peut pas être invoquée à l'extérieur de sa classe.
- Le langage Java n'a pas de pointeur arithmétique. Ainsi on ne peut pas laisser les pointeurs introduire des virus à l'intérieur de la mémoire. [28]

En plus d'éléments de sécurité du langage Java, Java Card présente un niveau plus élevé de sécurité, en étant un sous ensemble du langage Java. En Java Card certaines caractéristiques comme les threads, le garbage collection, le gestionnaire de sécurité, le chargement dynamique de classes, le clonage, la finalisation et le calcul des nombres réels sont retirés pour répondre à la préoccupation de sécurité et aux ressources limitées disponibles sur la carte. Le chargement dynamique de classes a toujours été une menace sérieuse dans Java car on pourrait charger des classes dont la JVM ne peut pas reconnaître jusqu'à ce qu'elles soient lancées. Ainsi, un programme qui alloue la mémoire en permanence conduira à des erreurs de mémoire qui peuvent endommager la carte. Java Card supporte les packages, le chargement dynamique des objets, les interfaces et les exceptions, etc. Ce qui ajoute plus de sécurité aux applications Java Card. [30]

4.3.2. Vérification du sous-ensemble des fichiers classes

Contrairement à la JVM, la JCVM a une architecture partagée en deux parties : le convertisseur, qui ne s'exécute pas sur la carte mais sur un ordinateur distant, et l'interpréteur, qui lui, s'exécute sur la carte. Le convertisseur est la partie terminale de la machine virtuelle et prend les fichiers .class en entrée. Durant la conversion les fichiers .class sont soumis au même niveau de vérification qu'ils le seraient par le vérifieur de la JVM.

Comme la technologie Java Card est un sous-ensemble du langage Java, le convertisseur doit vérifier également que les fichiers .class n'utilisent que les caractéristiques de ce sous-ensemble. Cette étape est appelée vérification du sous-ensemble (subset-checking). Durant cette étape le convertisseur vérifie la validité des règles : types non supportés (types long, double et float), threads, tableaux multidimensionnels, etc. [28]

4.3.3. Vérification du fichier CAP et du fichier d'exportation

Le fichier CAP est de même importance que les fichiers .class de la plate-forme Java. C'est un format binaire permettant de charger des packages Java sur une Java Card. En pratique on ne peut pas considérer la fiabilité d'un fichier CAP généré à partir de classes vérifiées comme acquise. C'est le rôle du vérifieur de fichier CAP de vérifier sa conformité.

La vérification a normalement lieu avant le chargement d'un programme dans la Java Card, cependant, en raison de son espace mémoire et capacités de calcul limités, il est difficile d'effectuer cette vérification sur la carte elle-même. Généralement, elle est effectuée en dehors de la carte à puce, juste avant le chargement de l'applet.

Le vérifieur s'assure que le fichier CAP a un format correct et qu'il répond à certaines contraintes comme la violation de mémoire, les types des champs, la visibilité ou non visibilité des variables et des méthodes. Durant la vérification le vérificateur s'assure que le fichier CAP est cohérent avec les fichiers d'exportation qu'il importe, et le fichier d'exportation qui représente son API. Ces fichiers contiennent les prototypes des classes importées et leurs méthodes, et sont produits lors de la génération du fichier CAP. Ainsi, la vérification des fichiers CAP nécessite non seulement les fichiers CAP mais aussi les fichiers d'exportation pour toutes les classes importées.

Quand la vérification du fichier CAP est terminée, le vérifieur attribue à l'applet une signature qui est vérifiée lors du chargement. Ce qui implique que le chargement d'applet doit être fait dans un environnement sécurisé ou en s'assurant de l'origine du code. La sécurité du système dépend alors de l'organisation qui signe le code, laquelle doit s'assurer que le code est bien typé.

Bien qu'il ne soit pas requis par les spécifications, certains fabricants de Java Card s'orientent vers la vérification des fichiers CAP sur carte, car cela permet un meilleur niveau de sécurité. [28][35]

4.3.4. Chargement d'applet

La spécification Java Card ne définit pas un processus de chargement, mais suggère qu'une applet dédiée (installateur) gère cette tâche.

Le Java Card Development Kit de SUN contient un simple installateur sans support cryptographique. Cet installateur peut être utilisé pour des buts de démonstration ou pour le chargement des applets de ROM, mais il n'est pas approprié pour le chargement des applets après-émission.

Global Platform définit une entité, située sur la carte, appelée *Gestionnaire de carte* (Card Manager) qui est capable d'effectuer un chargement plus sophistiqué. Généralement, le gestionnaire de carte met en œuvre un protocole de canal sécurisé qui fournit des services cryptographiques comme le chiffrement et l'authentification pour garantir l'intégrité et la confidentialité de l'applet lors du transfert et pour authentifier le serveur depuis lequel l'applet est chargée.

Dans la phase de chargement, il faut s'assurer aussi que le code de l'applet a bien été vérifié par le vérifieur de fichier CAP. L'applet possède une signature qui est vérifiée lors du chargement. Durant cette phase, il y a aussi une vérification de la place disponible sur la carte afin de garantir que l'applet pourra s'y exécuter. [1][35]

4.3.5. Pare-feu et contextes

Le gestionnaire de sécurité et le contrôleur d'accès de la plate-forme Java sont remplacés par le pare-feu (firewall) dans Java Card.

Une Java Card peut supporter plusieurs applets qui peuvent provenir de différents vendeurs, de sorte qu'il n'est pas sécurisé si les objets d'une applet sont accessibles à une autre applet. Le pare-feu des Java Cards est mis en œuvre dans le JCRE et permet de mettre en œuvre une politique d'isolation des applets et de contrôle d'interactions entre ces applets. Le pare-feu opère lors de chaque accès mémoire à l'exécution et impose une stricte séparation entre les espaces mémoire des différentes applets présentes sur la carte.

La politique d'isolation est basée sur la notion de « contexte ». Dans Java Card, il existe deux contextes. Le premier est le contexte simple, il est assigné à chaque instance d'applet, c.à.d. que chaque instance possède son propre contexte simple. Le deuxième est le contexte de groupe. Il regroupe un ensemble de contextes simples. Toutes les instances d'applet dont les classes sont déclarées dans le même package partagent le même contexte de groupe, c.à.d. que chaque package possède un contexte de groupe. Dans la suite, nous appelons contexte le contexte de groupe. Il existe un autre contexte de groupe, celui du JCRE.

Tous les objets ont un contexte, celui de leur propriétaire, c.à.d. celui de l'applet qui les a créés. Le propriétaire d'une applet est l'applet elle-même. Ainsi, l'exécution d'une méthode se fera toujours dans son contexte.

Le pare-feu sépare les contextes les uns des autres. Ainsi, les accès aux membres (champs et méthodes) entre deux instances qui ne sont pas dans le même package sont automatiquement refusés par le pare-feu. Les interactions entre deux instances d'un même package suivent seulement les règles de visibilité des champs et des méthodes. Si le pare-feu détecte un accès illégal, une exception de sécurité est levée et l'exécution s'arrête.

Néanmoins, il faut noter qu'il existe un mécanisme spécifique permettant malgré tout l'échange de données entre applications de contextes différents, il s'agit du protocole de partage SIO (Sharing Interface Object) qui spécifie comment les références sur des objets partagés peuvent être obtenues et quels sont les contrôles à effectuer à cet instant. [1][36]

4.3.6. Points d'entrée et tableaux globaux

La règle d'isolation des applets peut être trop contraignante dans certains cas. Il est inconcevable que les différents acteurs sur la carte ne puissent jamais s'échanger des informations. Par exemple, le JCRE reçoit des commandes via des APDUs et doit les transmettre aux applets. Or, d'après le principe d'isolation cela lui est impossible. Certains objets du JCRE ont donc un statut spécial. Il y a les points d'entrée : ce sont des objets dont les méthodes sont accessibles depuis n'importe quel contexte. Ces points d'entrée se divisent en deux catégories : les temporaires et les permanents. Les points d'entrée temporaires sont des objets qui ne sont créés que pour la session en cours d'exécution, ces objets seront détruits à la fin de la session. En raison de cette destruction, il est impossible de sauvegarder une référence sur ce type d'objet dans un champ. Cette restriction est forcée par le pare-feu, il refuse toutes les sauvegardes de ce type. Les points d'entrée permanents sont des objets dont la durée de vie est la même que le JCRE, il est donc possible de sauvegarder une référence sur de tels objets.

Certains tableaux appartenant au JCRE peuvent être globaux, c.à.d. que leur contenu est accessible depuis tous les contextes. Par exemple, le JCRE utilise un tableau d'APDU pour passer la commande à une applet. Ce tableau étant global, l'applet peut lire la commande que le CAD a envoyé. Le JCRE remet à zéro ce tableau lors des changements d'applets pour éviter qu'elles échangent des informations par ce biais.

Ces commandes sont passées aux applets avec l'utilisation des paramètres des méthodes *install*, *select*, *process* et *deselect*. Ces méthodes étant en dehors du contexte du JCRE, celui-ci ne devrait pas pouvoir les appeler. Mais le JCRE n'est pas soumis aux règles du pare-feu car il correspond au « système ». Il a donc accès à tout sans aucune restriction.

4.3.7. Partage d'objets

Les applets d'un même contexte peuvent échanger librement des informations. Par ailleurs, le JCRE peut communiquer avec les applets et les applets ont certains accès sur des objets du JCRE. Mais dans ce modèle, deux applets de deux contextes différents ne peuvent pas communiquer. Dans certains cas, ces applets peuvent avoir des informations à se communiquer. C'est dans ce but que Java Card définit les objets partagés. Un objet partagé est la propriété d'une applet, mais celle-ci autorise d'autres applets à appeler certaines méthodes de cet objet, sans que le pare-feu ne les bloque. [1]

4.3.8. Atomicité des transactions

Les cartes à puce fonctionnent souvent dans des environnements hostiles en dehors du contrôle de l'émetteur de la carte. Des conditions défavorables peuvent surgir en ce qui concerne les températures, les vibrations, l'humidité, etc. En outre, un détenteur de carte peut à tout moment retirer la carte du lecteur de carte. Ces circonstances peuvent parfois interrompre les opérations en cours.

Java Card introduit un mécanisme de transaction qui garantit son atomicité. Il permet de s'assurer que toutes les opérations ou aucune opération dans la transaction est terminée. Une opération de validation à la fin de la transaction confirme l'achèvement des opérations précédentes. Si la transaction est annulée (par une coupure de courant par exemple), le mécanisme permet de s'assurer que toutes les opérations précédentes au sein de la transaction sont inversées. De cette façon il est possible de maintenir la cohérence interne des données connexes.

Une conséquence du mécanisme de transaction est que les objets qui ont été alloués dans le cadre d'une opération avortée doivent être supprimés et leurs références sont réinitialisées. [35]

4.3.9. Cryptographie

Certaines APIs de Java Card permettent d'effectuer des opérations cryptographiques, comme le chiffrement, le déchiffrement et les signatures. Ces opérations peuvent être utilisées lors du stockage ou de l'échange d'informations sensibles pour garantir un certain niveau de sécurité. La cryptographie offre les fonctions de sécurité suivantes aux données :

- La confidentialité : les données ne peuvent-êtr comprises par une personne autre que le propriétaire.
- L'intégrité : Les modifications de données peuvent être détectées.
- L'authentification : la source et la destination de données peuvent être confirmées.
- La non-répudiation : l'émetteur de ne peut pas nier son message. [1] [30]

4.4. Obtenir un code mal typé sur une Java Card

Les attaques logiques consistent à exécuter des applications malveillantes, par exemple des applications pour lesquelles les instructions ne respectent pas les règles de typage de Java ou les règles de construction des fichiers d'entrée, ou encore utilisent des imprécisions d'une ou plusieurs des spécifications Java Card.

Nous allons présenter, ici, les trois façons les plus utilisées pour obtenir un code mal typé sur une Java Card : la manipulation des fichiers CAP, abusant des objets d'interface partageable, abuser le mécanisme de transaction. [36] [37]

4.4.1. Manipulation du fichier CAP

La façon la plus simple pour obtenir un code mal typé s'exécutant sur une Java Card est d'introduire dans un fichier CAP une faille de type dans le bytecode et de l'installer sur la carte. Bien sûr, cela ne fonctionnera que pour les cartes qui ne contiennent pas un vérifieur de bytecode sur carte et pour les fichiers CAP non-signés. [37]

L'édition de fichiers CAP donne des possibilités infinies pour produire des confusions de type dans le code.

E. Poll et W. Mostowski [37] proposent plusieurs attaques par confusion de type. L'objectif est de transformer un objet d'un type donné pour lequel un nombre limité d'opérations est possible en un objet pour lequel d'autres opérations deviennent éligibles. Par exemple, transformer un tableau d'octets en un tableau d'entiers permet de pouvoir lire deux fois plus de données et potentiellement des données sortant du domaine de l'applet. Une autre approche (d'abord proposée par Witteman [35] puis par Vertanen [38]) est de faire traiter à la machine virtuelle un objet en tant que tableau. Si les attributs du descripteur d'objet sont correctement renseignés et situés aux mêmes emplacements mémoire alors il devient possible de modifier la taille du tableau en tant qu'attribut légitime d'un objet. A partir de là, il est possible de l'utiliser en tant que tableau ayant une taille égale à la mémoire de la carte et il est possible de lire/écrire partout en mémoire à partir de l'adresse à laquelle est stocké l'objet. Les auteurs exploitent l'attaque précédente pour traiter une référence vers un objet en tant que short. De par ce transtypage, il est possible de lire et modifier des références existantes, chose théoriquement impossible sur une Java Card. Ils ont ainsi proposé plusieurs exploitations de cette méthode. Par exemple, il est possible d'échanger la référence de deux objets même si ceux-ci ont des types incompatibles. Un attaquant peut également manipuler l'AID associé à un objet et rendre invalide ainsi une applet dûment référencée par le système. De même, il devient possible de fabriquer des références et lire une partie de la mémoire. Néanmoins les auteurs expliquent qu'un certain nombre de contre mesures sont implémentées dans les cartes rendant cette attaque moins sensible. Hypponen dans sa thèse [39] suggère une autre approche. Il s'agit d'exploiter une faiblesse de l'utilisation du firewall avec les instructions manipulant des attributs globaux (getstatic et putstatic). L'instruction getstatic est employée pour lire le contenu d'un champ statique d'une classe. Les deux opérandes de cette instruction sont employés pour établir un index dans le constant pool. Pendant le chargement d'applet, le

processus d'édition de lien remplacera les deux opérandes par une adresse dans la mémoire. L'idée de l'attaque est de supprimer l'information qui demande la résolution de cette référence de sorte que cette dernière ne soit pas modifiée par l'édition de lien. Ceci est rendu possible de par l'utilisation d'un composant recensant les champs devant être résolus lors du chargement. Cette attaque fonctionne (en l'absence d'un vérifieur de bytecode) parce qu'aucun contrôle de contexte n'est fait par le firewall pendant l'accès au champ statique. Cependant, l'auteur n'a présenté aucun résultat expérimental ou même une implémentation de l'attaque. [36]

4.4.2. Abusant des objets d'interface partageable

Le mécanisme d'interface partageable de Java Card peut être utilisé pour créer de la confusion de type entre les applets sans aucune modification directe des fichiers CAP.

Les interfaces partageables permettent la communication entre les applets (ou entre les contextes de sécurité, pour être précis) : les références à des instances d'interfaces partageables peuvent être légalement utilisés à travers le pare-feu d'applet. Pour utiliser cette option pour créer une confusion type, on peut laisser deux applets communiquer via une interface partageable, mais compiler et générer des fichiers CAP pour des applets utilisant des définitions différentes de l'interface partageable. Ceci est possible parce que les applets sont compilés et chargés séparément.

4.4.3. Abuser le mécanisme de transaction

Le mécanisme de transaction de Java Card, est probablement l'aspect le plus délicat de la plate-forme Java Card. Le mécanisme a été démontré comme une ressource possible d'injections de fautes sur une seule carte. Le mécanisme de transaction permet aux instructions multiples de bytecode d'être transformées en une opération atomique, offrant un mécanisme de retour en arrière au cas où l'opération est abandonnée, ce qui peut arriver par une déchirure de carte ou par l'invocation de la méthode de l'API `JCSysystem.abortTransaction`. Le mécanisme de restauration doit également libérer tous les objets attribués au cours d'une transaction avortée, et de réinitialiser les références à de tels objets à null.

C'est ce dernier aspect qui peut être détourné pour créer de la confusion de type : si de telles références sont réparties autour, par exemple, les affectations à des champs et des variables locales, il devient difficile pour le mécanisme de transaction de garder la trace de quelles références devraient être réinitialisées à null.

Les implémentations boguées du mécanisme de transaction sur certaines cartes conservent la référence à un objet qui doit être réinitialisé et réinitialisent d'autres. Apparemment, ce potentiel de problèmes a été remarqué, comme la spécification du JCRE qui permet explicitement à une carte de couper dans le cas d'un avortement programmé, après que les objets ont été créés lors de la transaction. [37]

5. CONCLUSION

Aujourd'hui, une grande quantité de cartes à puces sont équipées avec une plate-forme Java Card. Elles sont largement utilisées dans les cartes SIM (utilisées dans les téléphones mobiles), les cartes de guichets automatiques, et les services financiers (cartes de crédit, porte-monnaie électroniques, etc.).

Ceci est grâce aux innovations majeures que Java Card a apporté dans le monde des cartes à puce. Parmi ces innovations, on trouve la possibilité de charger de nouvelles applications, appelées *applets*, sur la carte à puce après sa délivrance.

Cependant, la possibilité de charger de nouveau code sur la Java Card révèle de grands problèmes de sécurité. Il faut assurer que ces nouvelles applications ne menacent pas la sécurité de la carte en tant que plate-forme d'exécution. [5]

Nous avons présenté, dans ce chapitre, l'ensemble de mécanismes de sécurité fournis par la plate-forme Java Card pour faire face à ce problème. Ces mécanismes définissent des propriétés qui doivent être respectées par toute application afin d'assurer que son exécution ne risque pas de compromettre l'intégrité de la carte ou la confidentialité des autres données qui y sont stockées.

Un point clé de ces mécanismes de sécurité est la *vérification du fichier CAP*, qui permet de protéger la carte des attaques de confusion de type provenant de la manipulation du code contenu dans le fichier CAP.

La vérification d'applications est un domaine désormais bien connu. De nombreux vérificateurs existent pour différents langages. Leur but est toujours le même : effectuer des tests et statuer sur la correction de l'application. Comme le code contenu dans le fichier CAP est considéré comme du bytecode, sa vérification est inspirée de la vérification de bytecode Java.

Dans le chapitre suivant, nous allons présenter les différents aspects reliés à la vérification du bytecode Java et en suite du fichier CAP, ainsi que les différentes méthodes utilisées pour implémenter le vérifieur, vu les contraintes de puissance de calcul et d'espace mémoire présentes au niveau de la Java Card.

Chapitre V :

Vérification de fichier CAP

1. INTROCUCTION

La technologie Java Card offre à la carte à puce la possibilité de charger du code exécutable après son déploiement. Cependant, le schéma de déploiement n'est pas exempt de problème.

Le problème majeur identifié concerne la validité et l'innocuité du code chargé pour la machine virtuelle. La question est : comment se protéger d'un code potentiellement dangereux ? En effet, dans le schéma de chargement de Java Card, le convertisseur effectue deux tâches : la première consiste à vérifier le fichier *class* fourni en entrée, puis la deuxième convertit ce fichier *class* en fichier CAP. Mais par la suite, ce fichier CAP peut transiter sur un réseau (par exemple sur Internet) pour arriver au terminal effectuant le chargement dans la carte. Rien n'assure que le fichier CAP n'a pas été modifié et ne menace pas l'intégrité de la machine virtuelle. [40]

La façon la plus simple pour obtenir un code malveillant s'exécutant sur une Java Card est d'introduire une confusion de type dans le bytecode contenu dans un fichier CAP et d'installer ce fichier sur la carte. L'édition de fichiers CAP donne des possibilités infinies pour produire des confusions de type dans le code. L'objectif est de transformer un objet d'un type donné pour lequel un nombre limité d'opérations est possible en un objet pour lequel d'autres opérations deviennent éligibles. Par exemple, transformer un tableau d'octets en un tableau d'entiers permet de lire deux fois plus de données et potentiellement des données sortant du domaine de l'applet. [36]

Nous avons présenté, dans le chapitre précédent, les différentes attaques par confusion de type qui peuvent survenir de la manipulation du fichier CAP. La possibilité de la réalisation de telles attaques parvient du fait que le bytecode contenu dans le fichier CAP ne respecte pas les règles imposées sur le bytecode. Ces règles sont sous forme de contraintes statiques pour valider le format du fichier CAP et sous forme de contraintes structurelles pour valider sa syntaxe.

Le rôle de la vérification du fichier CAP est d'assurer que le bytecode contenu dans ce fichier respecte les contraintes statiques et dynamiques sur le bytecode. Durant la vérification le vérifieur s'assure aussi que le fichier CAP est cohérent avec les fichiers d'exportation qu'il importe, et le fichier d'exportation qui représente son API. [41]

Dans ce chapitre, nous allons tout d'abord présenter l'ensemble de la machine virtuelle Java en décrivant la structure du fichier class, les contraintes imposées sur les instructions du bytecode et le besoin d'un vérifieur de bytecode ainsi que sa place dans la JVM. Par la suite, nous allons présenter les différences majeures entre Java et Java Card au niveau de la machine virtuelle, du format des fichiers exécutables sur la machine virtuelle, et de la place et le comportement du vérifieur de bytecode. Et à la fin, nous allons présenter le vérifieur statique de fichier CAP hors-carte tel qu'il est décrit par la spécification de Sun, tout en décrivant la façon de procéder de ce vérifieur et les différentes vérifications qu'il effectue au niveau du fichier CAP et des fichiers d'exportation.

2. VUE D'ENSEMBLE DE LA JVM ET DU BYTECODE

2.1. Définition du bytecode

Le bytecode (code-octet) est un langage de programmation à pile de bas niveau. À l'origine, il a été conçu comme une représentation intermédiaire, normalement générée par un compilateur, entre du code source de haut niveau (principalement en langage Java) et un langage machine assembleur. Ce code est normalement destiné à être exécuté par une machine virtuelle et est distribué dans le format *class* sous forme de fichier ou de flux de données dans un réseau. Un fichier au format *class* représente une classe ou une interface en langage de haut niveau orienté objet. [42]

2.2. Format du fichier class

Un fichier *class* est encodé en une séquence de bytecodes. Un bytecode est représenté sur 8 bits. Les structures constituant le fichier *class* sont représentées avec des séquences de 1, 2 ou 4 octets. Les notations utilisées pour les représenter sont u1, u2, u4.

Un fichier *class* contient des informations comme la version du format de fichier utilisée, une table de constantes (*constant_pool*), la super-classe, les interfaces implémentées, les champs, les méthodes et les attributs de la classe ou de l'interface. Toute référence à une classe, une interface, une méthode ou un champ faite dans le fichier (par exemple la super-classe ou, dans le code d'une méthode, une instruction d'invocation de méthode) est exprimée comme une référence symbolique à une constante du *constant_pool*. Cette constante exprimera sous forme de chaîne de caractères le nom ou la description de l'item référé. Par exemple, l'item *super_class* d'une classe contient l'indice qui doit désigner une constante de type *CONSTANT_Class_info* dans la *constant_pool*. Cette constante indique le nom complet de la classe ; *java/lang/String* par exemple.

Les méthodes d'une classe sont encodées dans des structures `method_info`. Cela indique les propriétés de contrôle d'accès à la méthode (publique, privée, etc.), son nom, son descripteur et ses attributs. Le descripteur est représenté par une chaîne de caractères exprimant les paramètres entre parenthèses suivi du type de retour. Par exemple, le descripteur de la méthode « `int test (boolean b, Object o)` » est noté `(ZLjava/lang/Object;) I`. Nous distinguons le `Z` pour `boolean`, `Ljava/lang/ Object;` pour la référence à la classe `Object` et `I` pour le type de retour `int`.

Le code d'une méthode est représenté par une séquence de bytecodes dans l'attribut `code` de la méthode. [42]

Nous présentons dans la figure 5.1 la description du format du fichier `class`.

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods count;
    method_info methods[methods_count];
    u2 attributes count;
    attribute_info attributes[attributes_count];
}
```

Figure 5.1. Structure d'un fichier `class`. [41]

Nous nous intéressons plus particulièrement au tableau `methods` qui contient toutes les méthodes de la classe, qui sont enregistrées dans une structure du type `method_info`, dont la description est dans la figure 5.2.

```
method_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

Figure 5.2. Structure de method_info. [41]

La figure 5.3 présente l'attribut code qui se trouve dans les attributs de la structure method_info et qui contient les instructions du bytecode dans le tableau code[code_length].

```
Code_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 max_stack;
    u2 max_locals;
    u4 code_length;
    u1 code[code_length] ;
    u2 exception_table_length;
    { u2 start_pc;
      u2 end_pc;
      u2 handler_pc;
      u2 catch_type;
    } exception_table[exception_table_length];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

Figure 5.3. Structure de l'attribut code. [41]

La figure 5.4 montre un exemple simple de code source écrit en Java et le bytecode correspondant.

```
void spin() {
    int i;
    for (i = 0; i < 100; i++) {
        ;
    } }
}
```

Le compilateur peut compiler `spin()` à :

```
Method void spin()
0 iconst_0 // empiler la constante int 0
1 istore_1 // stocker dans la variable locale 1 (i = 0)
2 goto 8 // la première fois, ne pas augmenter
5 iinc 1 1 // augmenter la variable locale 1 par 1 (i++)
8 iload_1 // empiler la variable locale 1 (i)
9 bipush 100 // empiler la constante int 100
11 if_icmplt 5 // comparer et répéter si (i < 100)
14 return // retourner void quand fini
```

Figure 5.4. Exemple de code source Java et le bytecode correspondant. [43]

2.3. Bytecode et instructions d'une méthode

La machine virtuelle Java est une machine abstraite à base de pile d'exécution. La plupart des instructions dépilent (*pop*) leurs arguments de la pile, et empilent (*push*) leurs résultats sur la pile. De plus, un ensemble de registres (également appelés les variables locales) est fourni. Ils peuvent être accédés via des instructions `load` et `store` qui empile la valeur d'un registre donné sur la pile ou stocke le sommet de la pile dans le registre donné, respectivement. La pile et les registres font partie du rapport d'activation pour une méthode. Ainsi, ils sont préservés à travers des appels de méthode. Le point d'entrée d'une méthode indique le nombre de registres et de blocs de pile utilisés par la méthode, ainsi, permettant à un rapport d'activation de la bonne taille d'être alloué sur l'entrée de la méthode.

Dans le bytecode, le contrôle est manipulé par une variété d'instructions de branchement intra-méthode: le branchement sans conditions (`goto`), les branchements conditionnels (branche si le sommet de pile est 0), le branchement de multi-chemins (correspondant à `switch` au niveau de source de Java).

Dans la JVM, environ 200 instructions sont utilisées, y compris des opérations arithmétiques, les comparaisons, la création d'objet, les accès de champ et les invocations de méthode. [43]

L'exemple de la figure 5.5 montre le calcul de l'équation $a = b + c$ en utilisant une pile P et avec $b = 1$ et $c = 2$. Les opérandes b et c doivent d'abord être empilés. L'opération d'addition les dépile et empile le résultat. Cette valeur est ensuite dépilée pour être affectée à la variable locale a. [42]

Instruction		Variables locales				
		Pile		a	b	c
		0	1			
empiler(P, b)	avant :	-	-	-	1	2
	après :	1	-	-	1	2
empiler(P, c)	avant :	1	-	-	1	2
	après :	1	2	-	1	2
additionner	avant :	1	2	-	1	2
	après :	3	-	-	1	2
affecter(a, dépiler(P))	avant :	3	-	-	1	2
	après :	-	-	3	1	2

Figure 5.5. Additionner avec une machine à pile. [42]

2.4. Des instructions typées

Une caractéristique importante de la JVM est que la plupart des instructions sont typées, c'est-à-dire qu'elles sont spécialisées pour opérer sur un type (ou une famille de types) de données précis. Les types de données reconnus par la machine virtuelle sont répartis en **types primitifs** et en **types références**.

Les types primitifs désignent premièrement des nombres entiers représentés en complément à deux, ou des caractères : byte (8 bits signés), short (16 bits signés), int (32 bits signés), long (64 bits signés) et char (16 bits non-signés représentant des caractères Unicode). Le type boolean est aussi représenté sur 32 bits. Cela dit, les structures de la machine virtuelle étant en 32 bits, le type int généralise les types byte, short, char et boolean. Il y a aussi les nombres à virgule flottante : float (32 bits signés, simple précision) et double (64 bits signés, double précision). Enfin, le type returnAddress (32 bits) représente l'adresse relative d'une instruction dans le tableau de code ; ce type ne correspond cependant pas à un type de données de haut niveau du langage Java mais est généré par un compilateur pour supporter la construction de sous-routines (le concept de sous-routines n'est pas présenté ici car il n'est pas dans le cadre de notre étude).

Les types références représentent des structures dynamiques. Ce sont les classes, les interfaces et les tableaux. Ils comprennent : `initReference(C)`, `uninitReference(C, offset)`, `uninitThis`, `array(dim, type)` , `null`, `reference(r)`. Le type `null` est compatible avec toutes les références.

Le traitement des différents types de données est donc effectué par des familles d'instructions dont les membres sont spécialisés pour un type de donnée. Par exemple. En Java, une addition peut avoir comme opérande des valeurs de types primitifs numériques. Nous distinguons donc quatre instructions d'addition : `iadd`, `ladd`, `fadd` et `dadd`, qui opèrent respectivement sur des données de type entier (`int`), entier (`long`), virgule flottante simple précision (`float`) et virgule flottante double précision (`double`).

Reprenons l'exemple de la figure 5.5 pour implémenter l'équation $a = b + c$ en utilisant les instructions du bytecode. Nous représentons les variables locales `a`, `b` et `c` par les indices 0, 1 et 2. Nous remplaçons l'opération `empiler(P,b)` par l'instruction `iload_1`, l'opération `empiler(P,c)` par l'instruction `iload_2`, l'opération `additionner` par l'instruction `iadd` et l'opération `affecter(a,dépiler (P))` par l'instruction `istore_0`. La figure 5.6 montre cet exemple. [42][43]

Instruction		Pile		Variables locales		
		0	1	0	1	2
		<code>iload_1</code>	avant:	-	-	-
	après:	1	-	-	1	2
<code>iload_2</code>	avant:	1	-	-	1	2
	après:	1	2	-	1	2
<code>iadd</code>	avant:	1	2	-	1	2
	après:	3	-	-	1	2
<code>istore_0</code>	avant:	3	-	-	1	2
	après:	-	-	3	1	2

Figure 5.6. Additionner deux entiers avec le bytecode. [42]

2.5. Contraintes sur les instructions

La machine virtuelle Java impose certaines règles pour s'assurer que le fichier `class` est bien valide. Ces règles sont sous forme de contraintes statiques pour valider le format du fichier `class` et sous forme de contraintes structurelles pour valider la syntaxe du bytecode.

Les contraintes statiques concernent la disposition des bytecodes et aussi les relations entre les instructions et les opérandes. Par exemple, la première instruction doit être inscrite à l'adresse relative 0 du tableau de code ; ou encore, l'adresse relative en opérande d'une instruction effectuant un branchement doit désigner une instruction valide de la même méthode. [41] [42]

Les contraintes structurelles, aussi nommées contraintes dynamiques, concernent l'interaction des instructions entre elles. Pour assurer ses contraintes, le code doit respecter les conditions suivantes :

- *L'exactitude du type* : les arguments d'une instruction sont toujours des types attendus par l'instruction.
- *La taille de la pile* : une instruction ne dépile jamais un argument d'une pile vide, ni pousse un résultat sur une pile pleine (dont la taille est égale à la taille maximale de pile déclarée pour la méthode).
- *La position du compteur de programme* : le compteur de programme (CP) doit toujours pointer dans le code pour la méthode, au commencement d'une instruction valide (aucun ne doit tomber à la fin du code de méthode ; aucun branchement dans le milieu d'une instruction).
- *L'initialisation de registre* : un chargement d'un registre doit toujours suivre au moins un stockage dans ce registre ; en d'autre termes, les registres qui ne correspondent pas aux paramètres de méthode ne sont pas initialisés à l'entrée de méthode, et c'est une erreur de charger un registre non initialisé.
- *L'initialisation d'objet* : quand une instance d'une classe *C* est créée, une des méthodes d'initialisation pour la classe *C* (correspondant aux constructeurs de cette classe) doit être invoquée avant que l'instance de classe puisse être utilisée.
- *Le contrôle d'accès* : l'invocation de méthode, l'accès aux champs et aux références de classe doivent respecter les modificateurs de visibilité (*private*, *protected*, *public*, etc.) de la méthode, du champ ou de la classe. [43]

2.6. Bytecode et sécurité

Bien que les compilateurs respectent en général les contraintes imposées sur le bytecode, le processus de vérification est une mesure de sécurité pour palier à toute éventualité de corruption des données, que ce soit lors du stockage ou de la transmission sur un réseau, ou alors pour détecter d'éventuelles attaques envers la machine virtuelle ou d'autres applications avec des programmes intentionnellement mal formés. En effet, une classe peut être distribuée sur un réseau ouvert sans que l'on puisse s'assurer de l'authenticité du distributeur ou de l'intégrité des données. [42]

La vérification est un des principaux organes de sécurité de la JVM. Le processus de vérification a pour but premier d'assurer que le code d'une application respecte les contraintes statiques et (surtout) dynamiques présentées précédemment.

2.7. Approche de vérification

La vérification du code peut prendre plusieurs formes : une autorité supérieure (aussi appelée un tiers de confiance) peut donner son approbation sur le comportement du code d'une application. Elle certifie que celui-ci n'effectue aucune opération illégale ou interdite. Le code est validé par l'autorité qui met en jeu sa crédibilité. Pour cela, elle teste et vérifie l'application. Après cela, elle produit un certificat cryptographique identifiant l'application. Ce certificat est envoyé avec le code de l'application au client (qui va exécuter le code). Le client décrypte le certificat et s'assure qu'il correspond bien à l'application grâce aux protocoles cryptographiques. Une autre solution consiste à effectuer soi-même la vérification (effectuer la vérification au niveau de la JVM, ici), sans recourir à quiconque. Ici, le distinguo entre vérification statique et dynamique s'impose.

La vérification dynamique consiste à une approche défensive de la JVM qui peut garantir les contraintes sur le bytecode dynamiquement, au moment de l'exécution du code. En effet, il est possible de surveiller le code, de le surcharger par des tests qui préviennent d'une mauvaise utilisation et d'interrompre l'exécution si la menace devient trop sérieuse. Cependant, la vérification de ces conditions au temps d'exécution est chère et ralentit l'exécution de manière significative.

La vérification statique consiste à vérifier ces conditions une fois pour toutes, par l'analyse statique du code d'octet au chargement. Ici le programme est vérifié hors exécution. C'est un processus lourd et coûteux, mais comme il s'effectue au chargement, on en paye le prix une et une seule fois. C'est la solution la plus sécuritaire car, le code n'est pas encore exécuté et qu'il est déjà possible de savoir s'il est correct du point de vue de la sécurité. D'autre part, cette solution a l'avantage d'être beaucoup plus souple qu'une solution à de signature cryptographique car elle ne nécessite pas de structure de déploiement complexe.

La vérification statique est réputée comme étant un algorithme coûteux en temps d'exécution, et aussi en consommation mémoire. Cela peut pénaliser fortement le processus de chargement du code dans la machine virtuelle. Ainsi la vérification statique doit être exécutée par un tiers extérieur lorsque les contraintes matérielles sont jugées trop importantes (comme dans une carte à puce par exemple). [40] [42] [43]

La figure 5.7 montre la place qu'occupe le vérifieur statique dans son architecture. Il se place en amont du chargeur de code (non représenté sur la figure) et peut être mixé avec la phase d'édition de liens ceci par souci d'efficacité. En effet la vérification structurelle (que l'on présentera par la suite) est fortement liée à cette phase, et on peut ainsi réunir ces deux opérations. [44]

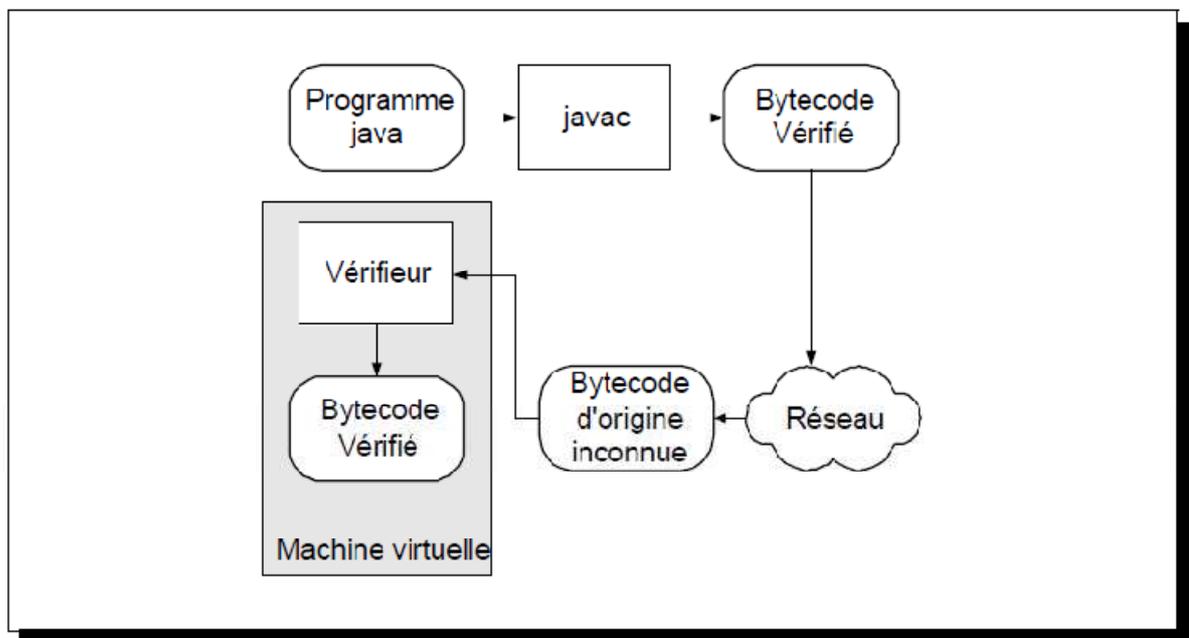


Figure 5.7. Place du vérifieur dans l'architecture de la JVM. [44]

3. DIFFERENCE ENTRE JAVA ET JAVA CARD

Nous allons, dans cette section, décrire les différences majeures existant entre la technologie Java et la technologie Java Card au niveau de la machine virtuelle, du format de fichiers exécutables sur la machine virtuelle, et de la place et le comportement du vérifieur de bytecode.

3.1. Machine virtuelle

La machine virtuelle Java Card comporte quelques différences par rapport à celle de Java. C'est une machine 16 bits. La taille élémentaire d'une cellule mémoire de la pile ou d'une variable locale est de 16 bits. De plus, les processeurs utilisés sur les cartes sont en grande partie des processeurs 8 bits (fondés sur des registres, des bus d'adresses, ou des bus de données de taille de 8 bits). La machine virtuelle Java Card ne va donc pas utiliser les types de taille supérieure à 32 bits car cela serait trop compliqué à gérer par le microprocesseur. Ainsi, le type le plus grand en Java Card est le type int. Certaines implémentations ne disposent même pas du type entier. De plus, les processeurs carte disposent rarement d'unité de calcul flottant. Ainsi, les types float et double ne sont pas gérés par la machine virtuelle Java Card. En Java Card nous ne disposons pas de tableau de dimensions multiples. De plus la taille des tableaux est limitée à 255. Enfin la JCVM ne dispose pas de thread, ceci en partie parce que les capacités de traitement de la carte ne le permettent pas.

3.2. Structure du fichier exécutable

Le format du fichier décrivant les classes n'est pas pensé de la même façon qu'en Java. Les Java Cards prennent en entrée des fichiers *.cap*. Ceux-ci sont issus de la conversion des *.class* de l'application à encarter. La différence majeure se situe au niveau du grain des applications qu'ils contiennent. Un *.class* Java contient le descriptif d'une et une seule classe. Un *.cap* Java Card contient le descriptif des classes d'un *package*. Ainsi, dans ceux-ci, il existe une distinction entre une classe dite *externe* (qui appartient à un autre package) et une classe *interne* (qui appartient au package courant).

Les *.cap* se décomposent en onze composants qui sont standardisés par SUN. Ils sont, dans l'ordre de chargement :

- Le *Header* qui contient des informations générales sur le package, comme par exemple le numéro de version de celui-ci et la version de la machine virtuelle qu'il requiert ;
- Le *Directory* qui décrit la taille de chacun des composants présents dans le package ;

- L'*Import* listant les dépendances avec les autres packages présents sur la carte ;
- L'*Applet* décrivant chaque applet du package ;
- Le *Class* contenant les informations sur toutes les classes et les interfaces (nous y trouvons par exemple les tables de méthodes virtuelles) ;
- Le *Method* qui contient les bytecodes de chaque méthode ;
- Le *StaticField* qui contient les informations pour créer et initialiser les classes du package (en particulier l'image statique de chaque classe) ;
- L'*Export* qui décrit chaque classe, méthode, champ publique.
- Le *ConstantPool* qui est un tableau permettant de retrouver les classes, les champs statiques et d'instances, ainsi que les méthodes ;
- Le *ReferenceLocation* qui facilite le travail de l'éditeur de liens ;
- Le *Descriptor* qui est essentiel à la vérification car il contient les informations de typage de chaque classe, champ, et méthode du package. [44]

3.3. Vérifieur de bytecode

La Java Card apporte à la carte à puce l'interopérabilité du langage Java. Ainsi, les applications développées pour Java Card peuvent s'exécuter sur toute carte implémentant la machine virtuelle Java Card. Le code Java Card généré est un code mobile qui peut transiter sur un réseau avant d'être chargé sur une carte à puce pour y être exécuté. La Java Card prévoit en effet la possibilité de charger du code après son déploiement. Cependant, le schéma de déploiement n'est pas exempt de problème.

Le problème majeur identifié concerne la validité et l'innocuité du code chargé pour la machine virtuelle. La question est : comment se protéger d'un code potentiellement dangereux ? En effet, dans le schéma de chargement de Java Card, le convertisseur effectue deux tâches : la première consiste à vérifier le fichier *Class* fourni en entrée, puis la deuxième convertit ce fichier *Class* en fichier CAP. Mais par la suite, ce fichier CAP peut transiter sur le réseau Internet pour arriver au terminal effectuant le chargement dans la carte. Rien n'assure que le fichier CAP n'a pas été modifié et ne menace pas l'intégrité de la machine virtuelle. Ainsi, la conformité du fichier CAP doit être vérifiée. [40]

L'application du modèle de Java à des dispositifs embarqués (tel que Java Card) exige que la vérification de bytecode soit effectuée sur le système embarqué, lui-même. Tout comme la machine virtuelle Java, la vérification effectuée par la JCVM peut être dynamique, au moment de l'exécution du code. Comme on a vu précédemment, ce processus est cher et ralentit l'exécution de manière significative (surtout pour les systèmes embarqués qui possèdent des ressources de stockage et de calcul limitées). Ou bien elle peut être faite statiquement par l'analyse statique du bytecode au chargement, ce qui est le rôle du vérifieur de fichier CAP.

Cependant, la vérification de bytecode est un processus cher qui peut excéder les ressources (capacité de traitement et espace mémoire) des petits systèmes embarqués. Par exemple, une Java Card typique a 1 ou 2 kilo-octets de mémoire de fonctionnement (RAM), 16 à 32 kilo-octets de mémoire persistante réenregistrable (*rewritable persistent memory*), et un microprocesseur de 8 bits qui est approximativement 1000 fois plus lent qu'un ordinateur individuel.

Avec de tels espaces mémoire et capacités de calcul limités, il est difficile d'effectuer cette vérification sur la carte elle-même. Généralement, elle est effectuée en dehors de la carte à puce, juste avant le chargement de l'applet.

Quand la vérification du fichier CAP est terminée, le vérifieur attribue à l'applet une signature qui est vérifiée lors du chargement. [28] [35]

- **Remarque**

Plusieurs solutions ont été proposées pour effectuer la vérification de bytecode sur les cartes à puces. Parmi ces solutions on trouve : l'approche proposée par le vérifieur pour cartes à puce de Trusted-Logic [45] qui consiste à la normalisation de bytecode avant son chargement. D'autres solutions ont été inspirées des travaux proposées et présentées par Necula et al dans [46], elles consistent à ajouter des preuves de la sûreté du programme au programme lui-même et de vérifier ces preuves au moment de chargement de programme sur la carte.

Ces techniques de vérification ne seront pas aborder dans ce mémoire, elles ne sont pas dans le cadre de notre étude. Dans la suite de ce chapitre, nous allons présenter le vérifieur statique de fichier CAP hors-carte tel qu'il est décrit par la spécification de Sun.

4. VERIFIEUR DE FICHIER CAP

Le rôle du vérifieur est de protéger la machine virtuelle contre tout débordement de l'application qui s'exécute. En particulier, il s'agit de vérifier que les instructions du programme respectent les contraintes statiques et dynamiques sur le bytecode.

Le vérifieur de fichier CAP s'assure que le fichier CAP a un format correct et qu'il répond à certaines contraintes comme la violation de mémoire, les types des champs, la visibilité ou non visibilité des variables et des méthodes. Il s'assure aussi que le fichier CAP est cohérent avec les fichiers d'exportation qu'il importe, et le fichier d'exportation qui représente son API. [28][35]

Dans cette section, nous allons présenter la description fournie par SUN du vérifieur de fichier CAP hors-carte, tout en décrivant son rôle et les différentes vérifications qu'il peut effectuer.

4.1. Description du vérifieur

Dans la plate-forme Java, la vérification statique d'une classe consiste à examiner le fichier class et l'ensemble des fichiers directement et indirectement référencés par cette classe. Dans la plate-forme Java Card, la vérification statique consiste à examiner le fichier CAP et l'ensemble des fichiers CAP directement et indirectement référencés via leurs fichiers d'exportation.

Les classes référencées et les fichiers CAP fournissent le contexte dans lequel une classe respective d'un fichier CAP va être exécutée. Prenons un exemple. Une méthode peut être examinée pour déterminer si elle est cohérente en vérifiant qu'elle ne contient pas une instruction de branchement qui tente de sauter en dehors de la méthode, ou ne tente pas d'effectuer une opération arithmétique sur un type de référence. Si la méthode appelle une autre méthode, les contrôles doivent être effectués pour déterminer si les paramètres passés lors de l'invocation correspondent aux paramètres attendus par la méthode appelée, entre autres choses. La méthode référencée, qui peut résider dans un fichier CAP différent, fournit le contexte approprié.

Le vérifieur hors carte de Sun supporte la vérification incrémentale et la résolution de l'ensemble de fichiers CAP qui seront installés sur la Java Card. L'unité de vérification est un fichier CAP unique. Le contexte dans lequel un fichier CAP peut être exécuté est assurée par l'API des packages référencés tels que définis dans leurs fichiers d'exportation. La résolution est validé hors carte en examinant les fichiers d'exportation des packages référencés.

Prenons un exemple d'un package d'une applet qui référence un package de la bibliothèque. Le fichier CAP du package de l'applet est vérifié et résolu en conjonction avec le fichier d'exportation du package de la bibliothèque. En outre, le fichier CAP de la bibliothèque est vérifié et résolu en conjonction avec son fichier d'exportation. Après que les deux fichiers CAP passent la vérification et la résolution avec le fichier d'exportation commun, ils sont considérés comme constituant un ensemble vérifié d'unités binaires.

La figure 5.8 montre un exemple plus détaillé de l'établissement d'un ensemble de fichiers CAP vérifié. Dans cet exemple, un package d'applet référence le package javacard.framework, et le package javacard.framework référence le package java.lang. Tout d'abord, le fichier CAP de java.lang est vérifié et résolu en conjonction avec le fichier d'exportation de java.lang. Ensuite, le fichier CAP de javacard.framework est vérifié et résolu en conjonction avec le fichier d'exportation de java.lang. Ceci établit que javacard.framework et java.lang constituent un ensemble vérifié. La vérification du fichier CAP du package de l'applet est faite de la même façon.

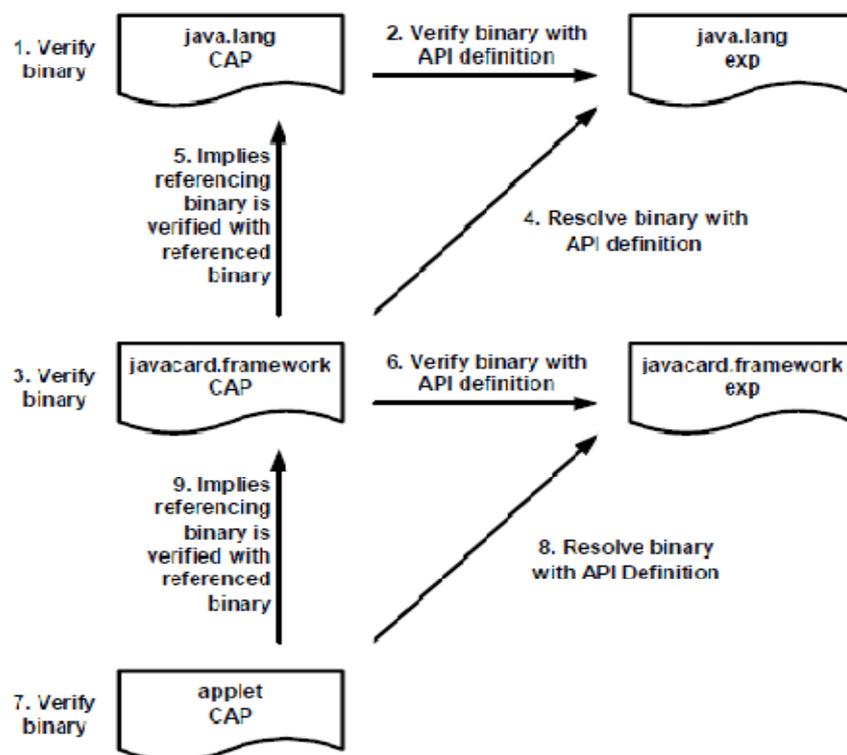


Figure 5.8. L'établissement d'un ensemble de fichiers CAP vérifié. [47]

Le processus de vérification statique de Sun commence avec les bibliothèques de bas niveau. Cela peut être caractérisé comme une vérification du bas en haut. Dans l'exemple de la figure 5.8, java.lang est vérifiée en premier lieu, et dans des vérifications ultérieures, java.lang n'est pas vérifié à nouveau.

Une Java Card est un environnement sécurisé. Des mesures de sécurité supplémentaires, telles que le pare-feu, évitent à une bibliothèque d'être corrompue. Une fois un fichier CAP vérifiée a été installé sur une Java Card son état ne peut pas être changé. Cela inclut à la fois son état interne et de son contexte.

On utilisant ce processus de vérification et de résolution hors-carte, on peut toujours affirmer que le contenu d'une Java Card est vérifié. Chaque bibliothèque qui a été placé sur la carte est vérifiée, et les installations suivantes sont résolues avec les définitions de l'API contenu dans carte. L'installateur sur carte de la Java Card accepte un fichier CAP seulement lorsque les fichiers CAP qu'il référence sont disponibles et ont des versions compatibles avec les versions des fichiers CAP utilisés lors de la préparation.

Un avantage important de la vérification statique incrémentale en utilisant les fichiers d'exportation est que les résultats sont indépendants de l'implémentation. Quand un fichier CAP du package référencé est résolu avec l'API de la bibliothèque, il est considéré comme résolu avec n'importe quelle implémentation de cette bibliothèque. Cette affirmation tient à condition que chaque implémentation de la bibliothèque ait été vérifiée avec le même fichier d'exportation. [47]

4.2. Processus de vérification

La figure 5.9 montre les entrées au vérifieur hors-carte lors de la vérification d'un fichier CAP. Il s'agit notamment du fichier CAP, le fichier d'exportation correspondant du fichier CAP, et les fichiers d'exportation des packages référencés par le fichier CAP. Si le fichier CAP n'export aucun objet, comme c'est le cas d'un package d'applet sans interfaces publiques partageables, alors le fichier d'exportation correspondant est omis. Le package java.lang est le seul cas où les fichiers d'exportation des packages référencés sont omis.

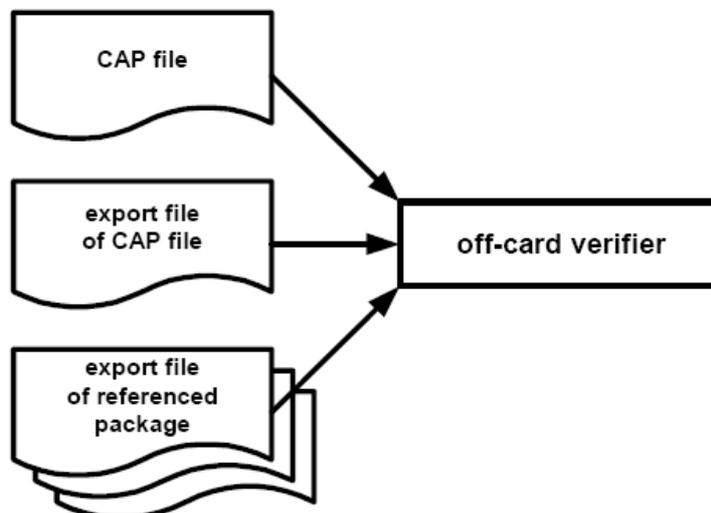


Figure 5.9. Vérification du fichier CAP. [47]

Le processus de vérification de fichiers CAP comprend trois ensembles de contrôles. Ces contrôles vérifient qu'un fichier CAP est :

- Intrinsèquement cohérent (cohérence interne).
- Cohérent avec son fichier d'exportation correspondant.
- Cohérent avec les fichiers d'exportation référencés.

Ces contrôles ne sont pas nécessairement exécutés séquentiellement. Chacun est décrit en termes généraux ci-dessous. [47]

4.2.1. Vérification de la cohérence interne

Il s'agit de la vérification du respect des contraintes statiques et dynamiques par le bytecode contenu dans le fichier CAP. Dans la vérification de bytecode d'un fichier CAP, nous distinguons deux types de vérifications :

- La vérification de structure ;
- La vérification de typage.

- ***La vérification de structure***

La vérification de structure consiste à vérifier que chaque *.cap* est correct d'un point de vue syntaxique. Le vérifieur s'assure qu'aucune des contraintes statiques sur le bytecode n'est violée.

Cette vérification concerne l'analyse du flot de données qui est reçu par la carte à puce. Lors du chargement, les composants constituant le fichier CAP sont envoyés l'un après l'autre afin d'être installés sur la carte. Cependant, il se peut tout à fait que la structure même des composants soit altérée et que certains éléments du composant soient alors inaccessibles.

L'analyse de la structure se base sur la définition de chaque composant par Sun [48]. La nature des tests comprend la taille de chaque élément, éventuellement leur valeur ou leur plage de valeurs. Bien entendu, nous ne pouvons présumer des tailles, mais nous nous attachons à vérifier la cohérence des informations. Ainsi, si l'élément *taille* d'un composant indique 10 octets mais que la somme des éléments de ce composant est différente, nous relevons une incohérence et nous rejetons le code.

La vérification de la structure se fait en deux étapes. La première est de vérifier la cohérence interne de chaque composant. C'est ce qu'on appelle les tests internes. L'idée est de s'assurer que les données qui sont censées représenter un composant ont les bonnes propriétés de ce composant. Pour établir les propriétés des composants nous nous servons de la description du fichier CAP fournie par Sun. Par exemple, nous savons que si nous vérifions le composant *Header* alors l'élément *tag* doit avoir la valeur 1 et l'élément *magic* doit avoir la valeur hexadécimale *DECAFFED*.

La deuxième étape de la vérification de structure consiste à vérifier les composants entre eux. Il arrive que deux composants se référencent mutuellement, ou qu'une donnée se trouve à la fois dans deux composants. Par exemple, le composant *Directory* contient un tableau appelé *component_size*. Ce tableau contient les tailles des différents composants du fichier CAP. Cet élément indiquant la taille est également défini pour chaque composant. Nous vérifions donc que les deux tailles indiquées sont égales. Si ce n'est pas le cas, nous refusons l'application. Ces vérifications sont nommées les tests externes.

Les attaques sur la structure sont les plus simples à faire. Pas besoin de rentrer au niveau des subtilités du typage de Java ou de faire des manipulations compliquées. Il s'agit assez simplement de modifier quelques valeurs dans le fichier CAP et de réajuster en conséquence les décalages pour avoir l'information désirée et créer ainsi une brèche de sécurité. En effet, si un code chargé n'a pas les bonnes propriétés, en indiquant par exemple une taille plus grande que ce qu'elle est réellement, alors, il y a une possibilité d'accéder à une plage mémoire interdite à l'application. Nous pouvons ainsi simplement violer l'une des règles de sécurité de Java Card qui est le confinement des applications. [40]

- ***La vérification de typage***

La vérification de typage consiste à une vérification des contraintes dynamiques sur le bytecode. Cette phase est la composante la plus complexe de la vérification. Son rôle principal est de s'assurer que chaque instruction qui compose un programme manipule les données (la mémoire) et déclenche l'exécution d'autres traitements en respectant les règles de typage définies par Java Card. [40]

Cette vérification est effectuée méthode par méthode et doit être faite pour chaque méthode du package, c'est-à-dire pour chaque méthode contenue dans le composant *Method* du fichier CAP à vérifier. Une description générale de cette vérification, mettant en évidence les points délicats, est donnée dans [49].

La partie vérification du typage assure qu'aucune conversion de type interdite d'après les règles de typage du langage Java Card n'est effectuée par le programme. Par exemple, un entier ne peut pas être converti en référence sur un objet, les conversions de type ne peuvent être effectuées qu'en ayant recours à l'instruction `checkcast` qui s'assure que le changement peut effectivement être effectué. De même, les arguments passés en paramètre à une méthode doivent être de types compatibles avec ceux attendus par la méthode.

Comme les types des variables locales ne sont pas explicitement stockés dans le bytecode, il est nécessaire de retrouver le type de ces variables en analysant le bytecode. Cette partie de la vérification est la plus compliquée, et la plus coûteuse à la fois en temps et en mémoire. En effet, cela nécessite de calculer le type de chaque variable et de chaque élément dans la pile pour chaque instruction et chaque chemin d'exécution possible.

Dans le but de rendre une telle vérification possible, la spécification fournie par Sun est assez restrictive sur les programmes qui sont acceptés. Seuls les programmes dont le type de chaque élément de la pile et de chaque variable locale est le même quelque soit le chemin pris pour atteindre cette instruction sont acceptés. Cela nécessite en particulier que la taille de la pile à une instruction donnée soit la même quel que soit le chemin menant à cette instruction.

[5]

4.2.2. Etapes de la vérification

Vérifier que le fichier CAP est intrinsèquement cohérent et cohérent avec le contexte des fichiers d'exportation référencés, peut être considéré comme un ensemble de vérifications qui sont énumérées ci-dessous.

1. Le chargement : Comporte la lecture d'un fichier CAP dans le vérifieur et de déterminer si tous les composants nécessaires du fichier CAP sont présents et que chacun est conforme au format général des composants tel que spécifié dans [48].
2. L'analyse : Comporte l'analyse de chaque élément pour confirmer que la syntaxe de base est correcte, et que les valeurs des champs individuels sont appropriées.
3. La liaison : Comporte la résolution des références inter- et intra-composants et confirme que les objets référencés sont compatibles avec les références.
4. Vérifier la sémantique du fichier CAP : Comporte la confirmation que les déclarations de classes et leurs hiérarchies sont compatibles avec les représentations de classes.
5. Vérifier les bytecodes : Consiste à une vérification de typage sur le bytecode de chaque méthode à fin d'assurer qu'aucune conversion de type interdite d'après les règles de typage du langage Java Card n'est effectuée par le programme.

Les références dans les étapes 3 à 5 sont validées à l'aide des informations de type fournies dans le composant *Descriptor* du fichier CAP. Les références à des éléments extérieurs sont en outre résolues en comparant leur utilisation et leurs informations de type dans le fichier CAP à leurs déclarations dans le fichier d'exportation du fichier CAP référencé.

Vérifier que le fichier CAP est cohérent avec son fichier d'exportation correspondant consiste à correspondre tous les éléments qui sont exportés à partir d'un fichier CAP avec celles déclarées dans son fichier d'exportation. Les deux ensembles doivent être exactement égaux. [47]

4.2.3. Vérification du fichier d'exportation

Chaque fichier d'exportation qui est fournie au vérifieur hors-carte au cours de la vérification du fichier CAP est également vérifié. Par ailleurs, le vérifieur hors-carte peut être invoqué pour vérifier un fichier d'exportation unique. La fonctionnalité dernière est fournie vue que les fichiers d'exportation jouent un rôle essentiel dans la construction des fichiers CAP par le convertisseur de Java Card et dans la vérification hors-carte.

La vérification d'un fichier d'exportation est peu profonde, ce qui signifie qu'un fichier d'exportation unique est examiné en un temps. L'ensemble des contrôles effectués confirment qu'il est intrinsèquement cohérent et en conformité avec la spécification de la JCVM. La vérification du contexte d'un fichier d'exportation est prise en charge indirectement lors de la vérification des fichiers CAP.

Les classes et interfaces représentées dans les fichiers d'exportation contiennent suffisamment d'informations sur leurs hiérarchies de construire un fichier CAP d'un package qui référence le package du fichier d'exportation. Quand une classe dans un fichier d'exportation étend une classe définie dans un autre package, certains éléments de la super-classe sont représentés. Ces éléments comprennent l'ensemble des super-classes, l'ensemble des méthodes virtuelles, et l'ensemble des interfaces implémentées. Chacun de ces ensembles doit être cohérent avec ceux figurant dans le fichier d'exportation de la super-classe. Si une incohérence existe entre les fichiers d'exportation, elle sera détectée lors de la vérification du fichier CAP vu qu'un fichier CAP doit être conforme à la fois avec le fichier d'exportation qu'il importe et le fichier d'exportation qu'il exporte. [47]

4.2.4. Vérification de la compatibilité

Les fichiers CAP peuvent dépendre des interfaces accessibles, classes, méthodes et champs d'autres fichiers CAP.

Dans un système largement distribué, un certain fichier CAP pourrait être utilisé par un certain nombre d'autres fichiers CAP. Ainsi, il est préférable que les différentes versions de ce fichier ne changent pas leur vue externe. En d'autres termes, toutes les classes, les interfaces, les méthodes et les champs accessibles publiquement, doivent toujours être accessibles de la même façon, sans aucune modification.

Le vérifieur hors-carte peut être invoqué pour vérifier la compatibilité entre deux versions d'un package en comparant les versions respectives des fichiers d'exportation. Cette vérification examine si les règles de version de Java Card ont été suivies. Cela inclut que les règles imposées à la compatibilité binaire tel que défini dans [48], ont été suivies. Le scénario est illustré à la figure 5.10.

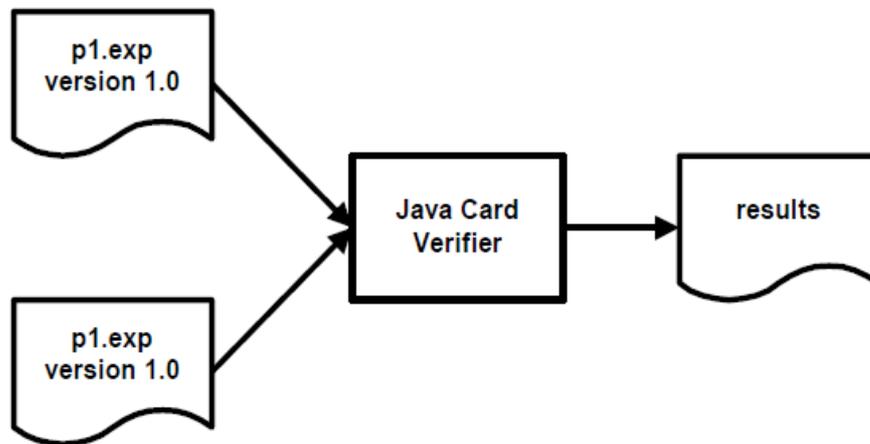


Figure 5.10. Vérification de la compatibilité. [47]

La comparaison des versions ne peut être faite qu'entre les packages qui ont le même numéro de version majeure. C'est parce qu'un changement dans le numéro de version majeure signifie qu'il ya eu un changement majeur dans la fonctionnalité. Un tel changement est considéré comme une incompatibilité binaire. Si les versions mineures des packages diffèrent par plus d'un point, cela peut signifier que les versions entre les deux ne sont pas compatibles. Par exemple, les versions 1.1 et 1.9 peuvent être compatibles au niveau binaire, mais 1.7 pourrait ne pas être compatible avec 1.9. Par conséquent, si les versions mineures diffèrent par plus d'un point, l'implémentation devrait soulever un avertissement que les versions entre les deux pourraient ne pas être compatibles. Cette incompatibilité est possible, même si les deux versions mineures sont vérifiées comme compatibles au niveau binaire. Si les deux versions mineures et majeurs des packages sont les mêmes, l'API doit être vérifié pour un contenu identique. [47]

4.2.5. Considérations sécuritaires

Comme la vérification est effectuée d'une façon incrémentale, il est nécessaire que tous les fournisseurs vérifient l'ensemble des fichiers CAP et d'exportation qu'ils produisent. Les consommateurs des fichiers CAP et des fichiers d'exportation peuvent ajouter de la redondance au processus en effectuant la vérification aussi.

La vérification hors-carte est destinée à être réalisée d'une manière sécurisée. Une fois un fichier CAP a passé la vérification, des mesures doivent être prises pour aider à assurer qu'il n'est pas corrompu avant qu'il ne soit installé sur une Java Card. Ces précautions sont dépendantes de l'implémentation et peuvent survenir du stockage du fichier CAP vérifiée dans un environnement de confiance, du cryptage du fichier CAP, de la signature du fichier CAP ou d'une autre mesure de sécurité.

Comme les fichiers d'exportation jouent un rôle essentiel dans les processus de conversion et de vérification hors-carte, les mesures de sécurité doivent être appliquées à eux aussi. Des précautions doivent être prises pour s'assurer qu'un fichier d'exportation n'est pas corrompu entre le moment où il est créé, vérifié avec son fichier CAP correspondant, utilisé par un convertisseur Java Card pour créer un fichier CAP de référencement, et vérifié avec ce fichier CAP de référencement. Si un fichier d'exportation est corrompu après la conversion d'un fichier CAP de référencement, la vérification du fichier CAP de référencement va échouer. [47]

5. CONCLUSION

Nous avons vu, dans ce chapitre, le besoin d'un vérifieur de fichier CAP avant son chargement dans la Java Card. Nous avons, en suite présenter le vérifieur de fichier CAP hors-carte de SUN. L'utilisation d'un tel vérifieur permet d'assurer que le code contenu dans le fichier CAP et qui va s'exécuter sur la carte ne menace pas la sécurité de la plate-forme et des applets présentes sur la carte. C.à.d. que tout fichier contenant une applet malveillante sera rejeté par le vérifieur.

Cependant, le vérifieur ne peut pas prévenir toutes les menaces possibles ; quelques codes malveillants peuvent passer la vérification sans que le vérifieur signale une erreur et rejette le code. Ceci est du à la présence de quelques vulnérabilités dans le système Java Card. Les applets contenant de tels codes peuvent, après leur installation sur la carte, effectuer des actions interdites, par exemple accéder à des zones de mémoire non autorisées et pouvant contenir des informations secrètes appartenant à d'autres applets.

Dans le chapitre suivant, nous allons présenter une vulnérabilité dans le système Java Card qui permet à un fichier CAP, contenant un code malveillant, de passer la vérification de fichier CAP hors-carte avec succès, et donc d'effectuer un certain type d'attaques sur la carte. Nous allons, ainsi, proposer et réaliser une contre mesure pour faire face à ce type d'attaques.

Chapitre VI :

Contribution

« Introduction d'un nouveau module de vérification dans le vérifieur de fichier CAP »

1. INTROCUCTION

Nous avons présenté, dans le chapitre précédent, la description fournie par SUN du vérifieur de fichier CAP hors-carte. Ce vérifieur est implémenté par les Java Cards qui utilisent la vérification de fichier CAP hors de la carte à puce.

Le rôle du vérifieur est de protéger la carte à puce et les applets présentes sur cette carte contre les attaques qui peuvent survenir du chargement de nouvelles applets sur cette carte. Il permet d'assurer que tout fichier CAP contenant une applet malveillante sera rejeté.

Ce vérifieur est très puissant et efficace contre les attaques de confusion de type. Cependant, on a réussi à trouver une faille dans le système Java Card qui permet d'introduire une attaque de confusion de type dans le code, que le vérifieur ne peut pas la détecter. Cette attaque a été présentée par les travaux de [6], ils ont réussi à exploiter une vulnérabilité présente dans le mécanisme de transaction offert par la plate-forme Java Card pour créer une attaque d'accès (en lecture et écriture) à la mémoire persistante complète d'un type particulier de Java Cards.

Dans ce mémoire, nous avons essayé d'introduire un nouveau module de vérification au vérifieur de fichier CAP hors-carte, qui permet de protéger la Java Card contre ce type d'attaques. Puisque nous ne possédons pas de carte réelle, nous avons utilisé un environnement de développement qui permet de simuler le type de cartes qui peuvent être attaquées par cette nouvelle attaque, et qui offre un outil de vérification hors-carte, afin de démontrer les résultats de notre travail.

Nous avons simulé une carte de crédit de type Java Card, développé et chargé une applet de porte monnaie électronique sur cette carte. Ensuite, nous avons exploité l'attaque présentée dans [6] pour développer une attaque sur cette applet. Pour faire cela, nous avons développé une nouvelle applet qui permet d'ajouter une nouvelle fonctionnalité à la carte (affichage des informations sur le titulaire, les dates, etc.) et nous avons introduit notre attaque dans cette applet.

Une fois que l'applet passe la vérification avec succès, elle peut être chargée sur la carte et attaquer l'applet de porte monnaie électronique. Le but de la réalisation de cette attaque est d'obtenir un gain financier ; elle permet à l'utilisateur de la carte de débiter des sommes de son compte même si son solde n'est pas suffisant.

Nous avons, ensuite, développé un nouveau module de vérification et introduit ce module dans le vérifieur fourni par l'environnement de développement que nous avons utilisé. Nous avons, par la suite, prouvé la capacité du nouveau vérifieur produit à détecter l'attaque précédente, afin d'empêcher le chargement de l'applet contenant cette attaque sur la carte.

Nous allons présenter, dans ce chapitre, le fonctionnement de l'environnement de développement Java Card que nous avons utilisé, et l'attaque décrite dans [6] qui permet l'accès à la mémoire persistante complète d'une Java Card et son exploitation dans notre travail. Ensuite, nous allons décrire le travail que nous avons réalisé, et qui consiste au développement et l'introduction d'un nouveau module de vérification dans l'ancien vérifieur. Nous allons, par la suite, présenter les résultats de l'utilisation du nouveau vérifieur qui démontrent son efficacité à détecter les différentes attaques de confusion de type. Nous terminons par une discussion des avantages et des inconvénients apportés par notre solution.

2. ENVIRONNEMENT DE DEVELOPPEMENT

La Technologie Java Card combine un sous-ensemble du langage de programmation Java avec un environnement d'exécution optimisé pour les cartes à puce et d'autres appareils à petites mémoires. L'objectif de la technologie Java Card est d'apporter de nombreux avantages du langage de programmation Java au monde de ressources limitées des cartes à puce.

L'API Java Card est compatible avec les normes internationales telles que ISO 7816, et de l'industrie des normes spécifiques tels que Europay, Master Card et Visa (EMV).

Le kit de développement pour la plate-forme Java Card (Javacard Development Kit 2.2.2) contient un ensemble de logiciels qui comprend les outils de simulation de Java Card, et un paquet de documentation qui comprend les livres pour l'utilisation de ces outils.

Avant d'utiliser ce kit, l'utilisateur doit se familiariser avec le langage de programmation Java, la conception orientée objet, les spécifications Java Card, et la technologie des cartes à puce. [50]

2.1. Description de l'environnement

Le kit de développement pour la plate-forme Java Card 2.2.2 (Javacard Development Kit 2.2.2) consiste à un ensemble d'outils de simulation pour la conception d'implémentations basées sur la technologie Java Card et le développement d'applets basées sur l'interface de programmation d'applications pour la plate-forme Java Card, version 2.2.2.

Toute implémentation d'un environnement d'exécution Java Card (JCRE) contient une machine virtuelle (VM) pour la plate-forme Java Card, les classes d'interface de programmation (API) Java Card, et des services de soutien.

Le kit de développement pour la plate-forme Java Card 2.2.2 est équipé avec un simulateur du JCRE, cref (C-language Java Card RE), qui est écrit dans le langage de programmation C et qui simule une implémentation d'une plate-forme Java Card de référence. Le cref implémente la spécification ISO 7816-4:2005, y compris le support pour un maximum de vingt canaux logiques, ainsi que les extensions des APDUs prolongés définis dans l'ISO 7816-3.

Il a été conçu pour simuler une implémentation de l'interface de communication d'une Java Card avec son environnement duale : $\mathbf{T} = \mathbf{1}$ (avec contact) et $\mathbf{T} = \mathbf{CL}$ (sans contact), avec la capacité de fonctionner sur les deux interfaces simultanément. [50]

2.2. Ensemble d'outils

Les logiciels inclus dans le kit de développement pour la plate-forme Java Card 2.2.2 contiennent toutes les spécifications Java Card, l'interface de programmation d'application pour la plate-forme Java Card, version 2.2.2, la spécification de l'environnement d'exécution pour la plate-forme Java Card, version 2.2.2, et la spécification de la machine virtuelle pour la plate-forme Java Card, version 2.2.2. Dans cette section, nous allons décrire l'ensemble d'outils contenus dans ce kit. [50]

2.2.1. L'outil « jcwde » (Java Card Workstation Development Environment)

L'environnement de développement et d'exécution de la plate-forme Java Card jcwde est un outil qui permet la simulation d'exécution d'une applet Java Card comme si elle était masquée dans la ROM. Il émule l'environnement de la carte.

Le jcwde n'est pas une implémentation de la machine virtuelle Java Card. Il utilise la machine virtuelle Java pour émuler l'environnement d'exécution Java Card. Pour la version 2.2.2 de l'implémentation de référence de la plate-forme Java Card, le jcwde ajoute le support pour l'invocation de méthode à distance (Java Card RMI).

Les fonctionnalités du JCRE qui ne sont pas prises en charge par le jcwde sont les suivantes:

- L'installation du package.
- L'état de la carte persistante.
- Le pare-feu.
- Les transactions.
- Le nettoyage du tableau transitoire.
- La suppression des objets.

- La suppression des applets.
- La suppression des packages. [50]

2.2.2. L'outil « cref »

L'implémentation de référence pour la plate-forme Java Card est écrite dans le langage de programmation C et est appelée cref ("C-langage Java Card RE"). Il s'agit d'un simulateur de l'environnement d'exécution Java Card (JCRE) qui peut être construit avec un masque de la ROM, un peu comme une implémentation d'une carte réelle basée sur la technologie Java Card. Il a la capacité de simuler la mémoire persistante (EEPROM), et de sauvegarder et restaurer le contenu de l'EEPROM envers et à partir des fichiers de disque. Les applets peuvent être installées dans le cref. Le cref effectue des E/S via une interface de socket, simulant l'interaction avec un lecteur de carte implémentant les communications T = 1, T = CL, ou T = 0 avec le lecteur de carte (CAD). Il supporte :

- L'utilisation de jusqu'à trois canaux logiques.
- Le type de données entier.
- La suppression d'objet.
- La réinitialisation de la carte en cas d'allocation d'objet au cours d'une transaction avortée. [50]

2.2.3. L'outil « apdutool »

L'outil apdutool est utilisé pour simuler l'interface de communication entre une carte à puce Java Card et le lecteur de la carte (CAD). Il lit un fichier de script contenant des commandes APDUs et envoie ces commandes au cref ou bien au jcwde. Chaque APDU est traitée et retournée à l'apdutool, qui affiche à la fois la commande et la réponse APDU sur le console. Optionnellement, l'apdutool peut écrire ces informations dans un fichier journal. [50]

2.2.4. L'outil « Converter »

L'outil Converter est utilisé pour transformer les fichiers class qui composent un package Java.

En plus des fichiers class, le convertisseur peut traiter soit la version 2.2.x, ou bien la version 2.1.x des fichiers d'exportation. Selon les options de ligne de commande, le convertisseur génère un fichier CAP, un fichier JCA, et un fichier d'exportation.

Le fichier CAP est un fichier au format JAR qui contient la représentation binaire exécutable des classes dans un package Java. Le fichier CAP contient également un fichier de type *manifest* qui fournit des informations lisibles sur le package que le fichier CAP représente.

Notons que le vérifieur effectue, avant la transformation, une vérification du sous-ensemble des fichiers classes, en suivant les limitations du langage Java Card définies dans [51]. [50]

2.2.5. L'outil « **offcardverifier** »

La vérification hors-carte fournit un moyen d'évaluer les fichiers CAP et les fichiers d'exportation dans un simple ordinateur. Lorsqu'elle est appliquée à l'ensemble de fichiers CAP qui vont être chargés sur une Java Card, et l'ensemble de fichiers d'exportation utilisés pour construire ces fichiers CAP, le vérifieur hors-carte (**offcardverifier**) fournit le moyen d'affirmer que le contenu de la carte à puce a été vérifié.

Le vérifieur hors-carte offert par le kit de développement pour la plate-forme Java Card 2.2.2 (Javacard Development Kit 2.2.2) est une implémentation de l'algorithme de vérification hors-carte de Sun que nous avons présenté dans le chapitre précédent. Il consiste à une combinaison de trois outils : `verifycap`, `verifyexp`, et `verifyrev`, dont :

- « **verifycap** » : L'outil `verifycap` est utilisé pour vérifier un fichier CAP dans le contexte de fichier d'exportation de son package (s'il y en a un) et les fichiers d'exportation des packages importés. Cette vérification confirme si un fichier CAP est intrinsèquement cohérent, telle que défini dans le chapitre précédent, et compatible avec un contexte dans lequel il peut résider dans une Java Card.
- « **verifyexp** » : L'outil `verifyexp` est utilisé pour vérifier un fichier d'exportation comme une seule unité. Cette vérification est «superficielle», elle examine seulement le contenu d'un fichier d'exportation unique, et n'incluse pas les fichiers d'exportation référencés par le package du fichier d'exportation. La vérification détermine si un fichier d'exportation est intrinsèquement cohérent et viable tel que défini dans [51].
- « **verifyrev** » : L'outil `verifyrev` est utilisé pour vérifier la compatibilité binaire entre les versions d'un package en comparant les fichiers d'exportation respectifs. La vérification examine si les règles de la version de la plate-forme Java Card, y compris celles qui sont imposées pour assurer la compatibilité binaire tel que définies dans [51], ont été suivies. [50]

2.2.6. L'outil « scriptgen »

L'outil scriptgen est utilisé pour simuler le programme d'installation hors-carte. Il permet de convertir un package contenu dans un fichier CAP en un fichier de script. Le fichier script contient une séquence d'APDUs au format ASCII approprié pour un autre outil, comme l'apdutool, pour l'envoyer au CAD à fin d'être chargé sur le simulateur de la plate-forme Java Card cref ou bien jcwde. L'ordre des composants du fichier CAP dans le script APDU est identique à l'ordre recommandé par la spécification de la machine virtuelle pour la plate-forme Java Card, version 2.2.2 [48]. [50]

2.2.7. L'outil « Installer »

L'Installer est l'outil offert par le kit de développement pour la plate-forme Java Card 2.2.2 qui représente l'installateur sur carte. Il permet de charger un package Java Card au format d'un script APDU dans une carte à puce et peut aussi le supprimer, ainsi que les applets. L'installer peut être utilisé pour:

- Charger dynamiquement un package sur une carte à puce de type Java Card. Au cours du développement, le fichier CAP peut être installé sur le simulateur du JCRE cref, plutôt que sur une carte réelle. L'installateur est capable de charger les versions 2.1, 2.2, 2.2.1, 2.2.2 des fichiers CAP.
- Effectuer au besoin, l'édition de liens sur la carte.
- Supprimer des applets et des packages d'une Java Card. Une fois l'installateur est sélectionné, les demandes de suppression peuvent être envoyés, à partir du terminal, à la carte sous forme de commandes APDUs.
- Le réglage, par défaut, des applets sur les différents canaux logiques.

L'installer n'est pas une application multiselectable. Au démarrage, l'installer est l'applet par défaut sur le canal logique 0. L'applet par défaut sur les autres canaux logiques est définie à *No applet selected*. [50]

2.2.8. L'outil « capgen »

Avant de décrire la fonction de l'outil capgen, nous allons, en premier définir ce qu'est un fichier JCA.

- **Fichier JCA (Java Card Assembly)** : C'est une représentation textuelle lisible d'un fichier CAP qu'on peut utiliser pour faciliter les tests et le débogage.

L'outil capgen est utilisé pour générer un fichier CAP à partir d'un fichier JCA (Java Card Assembly) donné. Le fichier CAP qui est généré a le même contenu d'un fichier CAP produit par le convertisseur. L'outil capgen est un back-end au convertisseur. [50]

2.2.9. L'outil « capdump »

L'outil capdump est utilisé pour créer une version ASCII (ensemble de commandes APDUs) d'un fichier CAP pour faciliter le débogage. [50]

2.2.10. L'outil « exp2text »

C'est un outil qui permet de visualiser n'importe quel fichier d'exportation en format texte. [50]

2.3. Ensemble de transformations de fichiers

Un fichier source écrit en Java peut être converti en un ensemble de commandes APDUs pour être utilisé sur une carte à puce de type Java Card (ou bien sur un simulateur de la carte). Le flux de données commence par la compilation du programme source écrit en Java, ensuite sa transformation par le convertisseur. L'outil Converter peut convertir les classes qui composent un package Java en un fichier CAP ou en un fichier JCA.

Un fichier JCA peut être utilisé comme une entrée à l'outil capgen pour créer un fichier CAP.

Le fichier CAP est vérifié par l'outil verifycap qui prend en entrée le fichier CAP, les fichiers d'exportation des packages importés, et optionnellement, le fichier d'exportation de son package. Le vérifieur vérifie la cohérence interne du fichier CAP, s'il trouve une anomalie (une violation des contraintes statiques ou dynamiques sur le bytecode, vues dans le chapitre précédent) dans le code contenu dans le fichier, il arrête le processus de vérification et affiche l'erreur détectée. Si le vérifieur ne trouve aucune anomalie dans le code, la vérification se termine avec succès et en résultat, il produit un fichier CAP vérifié.

▪ Remarque

Généralement, dans le cas d'une Java Card réelle, le fichier CAP vérifié contient une signature numérique, accordée par le vérifieur, et vérifiée par l'installateur, lors du processus de chargement du fichier sur la carte. Le vérifieur offert par notre simulateur (l'outil verifycap) ne supporte pas cette fonctionnalité. C.à.d. que le vérifieur n'accorde pas une signature numérique au fichier CAP vérifié, il affiche simplement si le code contenu dans le fichier contient des erreurs ou pas. C'est au développeur, ensuite, de prendre le fichier vérifié et de le charger sur la carte à travers les outils offerts par le kit.

Après la vérification, le fichier CAP est traité par l'installateur hors-carte scriptgen. Cela produit un fichier script d'APDUs, qui est utilisé comme une entrée pour l'outil apdutool. L'apdutool envoie l'ensemble d'APDUs à l'implémentation de l'environnement d'exécution Java Card (le cref).

L'utilisation des autres outils offerts par le kit est optionnelle. L'utilisateur peut utiliser n'importe quel outil en fonction de ces besoins. Par exemple il peut invoquer l'outil verifyrev pour vérifier la compatibilité binaire entre deux versions d'un package s'il en a besoin d'effectuer cette vérification pour son développement.

La figure 6.1 montre l'ensemble de transformations effectuées pour les différents types de fichiers utilisés pour une plate-forme Java Card. Les outils capdupm, exp2text, Installer, offcardverifier, et jcwde ne sont pas montrés sur la figure. [50]

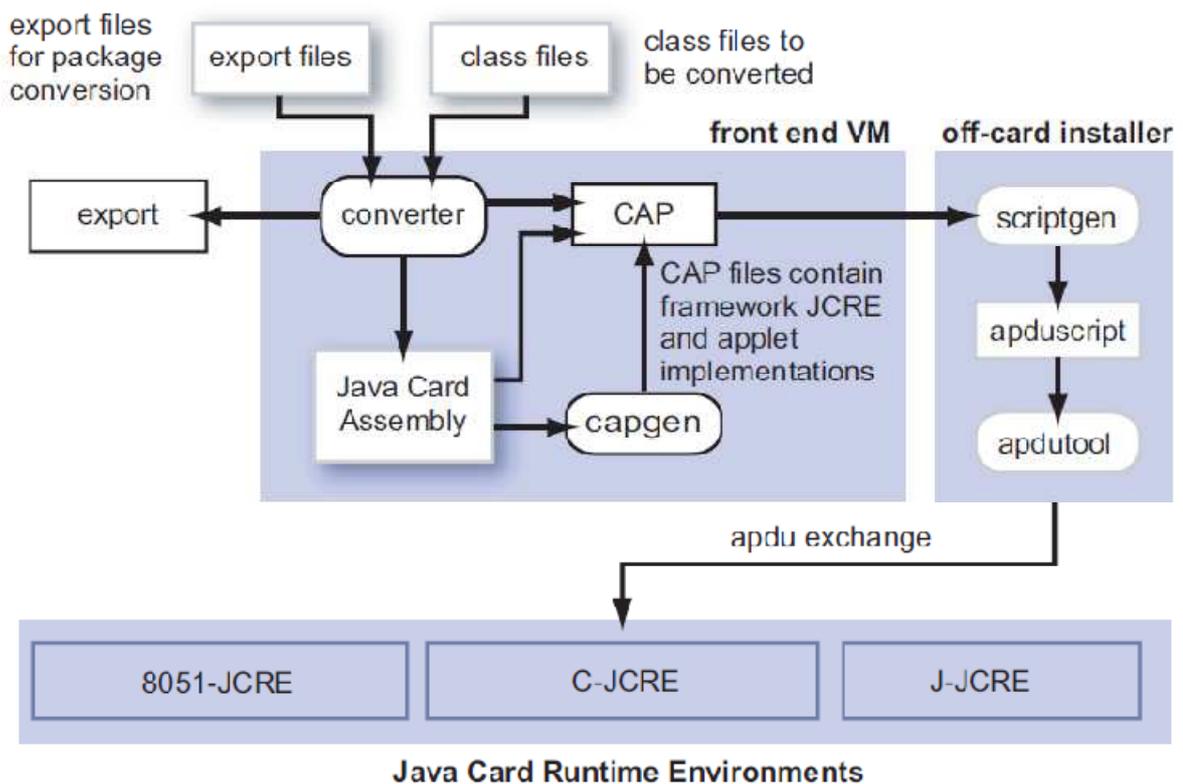


Figure 6.1. Ensemble de transformations de fichiers pour Java Card. [50]

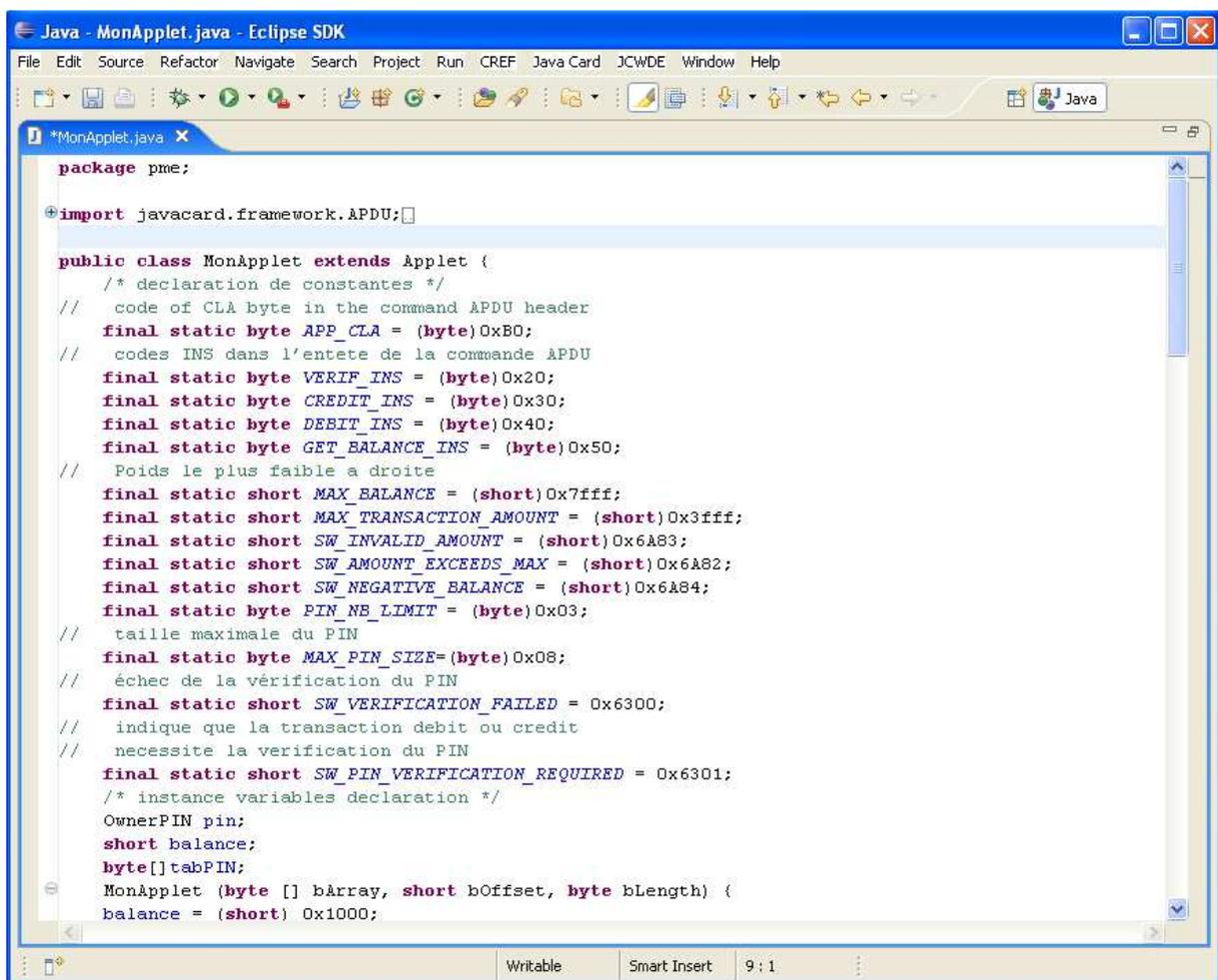
2.4. Exemple d'utilisation du kit

Le développeur d'applications pour une carte à puce de type Java Card, peut utiliser les différents outils de simulation offerts par le kit de développement pour la plate-forme Java Card 2.2.2 (Javacard Development Kit 2.2.2), qui lui permettent de développer des applets Java Card et des applications clientes, d'effectuer les différents transformations nécessaires des programmes, et de simuler leurs chargement sur la carte et leur exécution.

Le kit est contenu dans une archive (appelée JCDK). Pour pouvoir l'utiliser, il suffit de dézipper cette archive et de définir 2 variables d'environnement : **JAVA_HOME** contenant le chemin d'accès à Java et **JC_HOME** contenant le chemin d'accès au JCDK. La définition de ces variables va nous permettre d'utiliser les outils de l'environnement en ligne de commande.

Dans cette section nous, allons illustrer l'utilisation du kit de développement par un exemple. Nous allons écrire une applet Java Card qui permet la gestion d'un porte monnaie électronique (utilisée dans une carte de crédit), effectuer les différents transformations et vérifications sur l'applet, la charger sur le simulateur du JCDK, et l'exécuter et interagir avec elle à travers l'interface de communication. Nous allons, par la suite, démontrer l'utilisation d'une application cliente.

2.4.1. Développement et compilation de l'applet



```
Java - MonApplet.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run CREF Java Card JCWDE Window Help

*MonApplet.java x
package pme;

import javacard.framework.APDU;

public class MonApplet extends Applet {
    /* declaration de constantes */
    // code of CLA byte in the command APDU header
    final static byte APP_CLA = (byte)0xB0;
    // codes INS dans l'entete de la commande APDU
    final static byte VERIF_INS = (byte)0x20;
    final static byte CREDIT_INS = (byte)0x30;
    final static byte DEBIT_INS = (byte)0x40;
    final static byte GET_BALANCE_INS = (byte)0x50;
    // Poids le plus faible a droite
    final static short MAX_BALANCE = (short)0x7fff;
    final static short MAX_TRANSACTION_AMOUNT = (short)0x3fff;
    final static short SW_INVALID_AMOUNT = (short)0x6A83;
    final static short SW_AMOUNT_EXCEEDS_MAX = (short)0x6A82;
    final static short SW_NEGATIVE_BALANCE = (short)0x6A84;
    final static byte PIN_NE_LIMIT = (byte)0x03;
    // taille maximale du PIN
    final static byte MAX_PIN_SIZE=(byte)0x08;
    // échec de la vérification du PIN
    final static short SW_VERIFICATION_FAILED = 0x6300;
    // indique que la transaction debit ou credit
    // necessite la verification du PIN
    final static short SW_PIN_VERIFICATION_REQUIRED = 0x6301;
    /* instance variables declaration */
    OwnerPIN pin;
    short balance;
    byte[] tabPIN;
    MonApplet (byte [] bArray, short bOffset, byte bLength) {
        balance = (short) 0x1000;
    }
}
```

Figure 6.2. Code de l'applet MonApplet.

Le code de l'applet est écrit en langage Java, il est compilé et enregistré dans un fichier .class. La structure d'une applet Java Card est définie dans le Chapitre 4. L'applet que nous allons développer, ici, est appelée **MonApplet** et elle est contenue dans un package appelé **pme**. La figure 6.2 montre un échantillon du code de l'applet écrit en Java dans l'environnement de développement Eclipse.

2.4.2. Conversion en fichier CAP

La première transformation effectuée est la conversion des fichiers class contenant les différents applets (dans notre cas, un seul fichier) en un fichier CAP contenant le package entier. La conversion est effectuée en utilisant l'outil Converter.

La figure 6.3 illustre l'utilisation de l'outil Converter en ligne de commandes, tout en montrant les paramètres passés à l'outil et le résultat de la conversion.

```

Microsoft Windows XP [version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

G:\Documents and Settings\yousra>setvars
G:\Documents and Settings\yousra>cd workspace
G:\Documents and Settings\yousra\workspace>cd Porte Monnaie
G:\Documents and Settings\yousra\workspace\Porte Monnaie>cd bin
G:\Documents and Settings\yousra\workspace\Porte Monnaie\bin>converter -exportpa
th "G:\JCDK\java_card_kit-2_2_2\api_export_files" -applet 0X01:0X02:0X03:0X04:0X
05:0X06:0X07:0X08:0X09:0X00:0X00 MonApplet pme 0X01:0X02:0X03:0X04:0X05:0X06:0X0
7:0X08:0X09:0X00 1.0

Java Card 2.2.2 Class File Converter, Version 1.3
Copyright 2005 Sun Microsystems, Inc. All rights reserved. Use is subject to lic
ense terms.

conversion completed with 0 errors and 0 warnings.
G:\Documents and Settings\yousra\workspace\Porte Monnaie\bin>_

```

Figure 6.3. Utilisation de l'outil Converter.

Si la conversion se déroule normalement, le convertisseur crée un dossier appelé **Javacard** qui contient 2 fichiers : **pme.cap** (le fichier CAP) et **pme.exp** (le fichier d'exportation correspondant au fichier CAP). Le convertisseur peut créer, optionnellement, un fichier **pme.jca**.

2.4.3. Vérification du fichier CAP

On vérifie le contenu du fichier CAP, produit dans l'étape précédente, à travers l'outil `verifycap`. La fonction de vérification, dans le simulateur, est limitée à afficher les erreurs trouvées dans le code s'il en contient, ou bien d'afficher qu'il ne contient pas d'erreurs. La figure 6.4 illustre l'utilisation du vérifieur pour notre applet.

```

C:\ Invite de commandes
G:\Documents and Settings\yousra>
G:\Documents and Settings\yousra>
G:\Documents and Settings\yousra>verifycap "G:\JCDK\java_card_kit-2_2_2\api_export_files\javacard\framework\javacard\framework.exp" "G:\JCDK\java_card_kit-2_2_2\api_export_files\javacard\crypto\javacard\crypto.exp" "G:\JCDK\java_card_kit-2_2_2\api_export_files\javacard\security\javacard\security.exp" "workspace\Porte Monnaie\bin\pme\javacard\pme.cap"
1.3 Off-Card Verifier, Version {1}.
Copyright 2005 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms.
Vérification du fichier CAP workspace\Porte Monnaie\bin\pme\javacard\pme.cap
La vérification est terminée, 0 avertissements et 0 erreurs.
G:\Documents and Settings\yousra>_

```

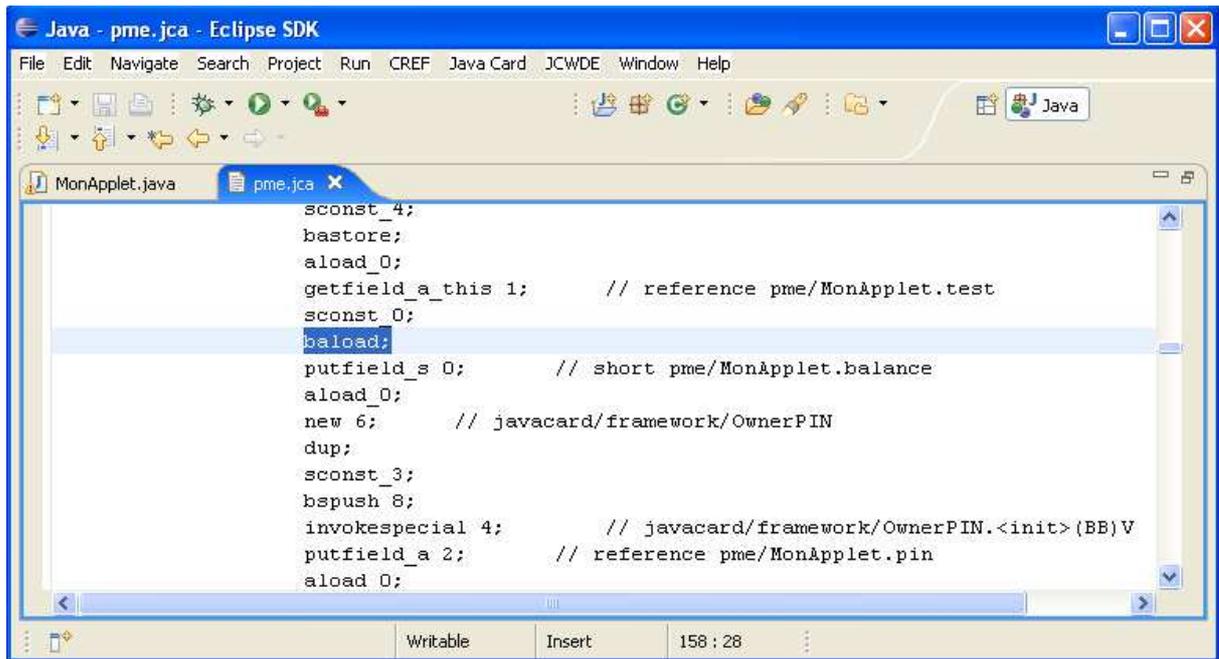
Figure 6.4. Utilisation de l'outil verifycap.

Dans cet exemple le fichier **pme.cap** ne contient aucune erreur. Nous allons, maintenant, montrer le résultat de la vérification du fichier si le code contient une erreur de confusion de type.

La confusion de type est normalement introduite dans le code de l'applet contenu dans le fichier CAP. Cependant, comme le format du fichier CAP est illisible par l'utilisateur, nous allons introduire le code qui génère la confusion de type dans le fichier **pme.jca**, ensuite, convertir ce fichier en un fichier CAP.

Nous allons introduire une simple confusion de type. Il suffit de définir un tableau de type byte et lire un champ de ce tableau en tant qu'une variable de type short. La taille du type short est le double de la taille du type byte dans la mémoire. Ceci va nous permettre la lecture de données contenues dans la zone mémoire qui suit celle de la variable lue, et qui peuvent être des données confidentielles ou qu'on n'a pas le droit d'y accéder.

La figure 6.5 montre un échantillon du code contenu dans le fichier **pme.jca**. L'instruction de lecture du champ d'un type byte est sélectionnée. Il suffit de remplacer cette instruction (`baload ;`) par l'instruction (`saload;`) pour obtenir la confusion de type.



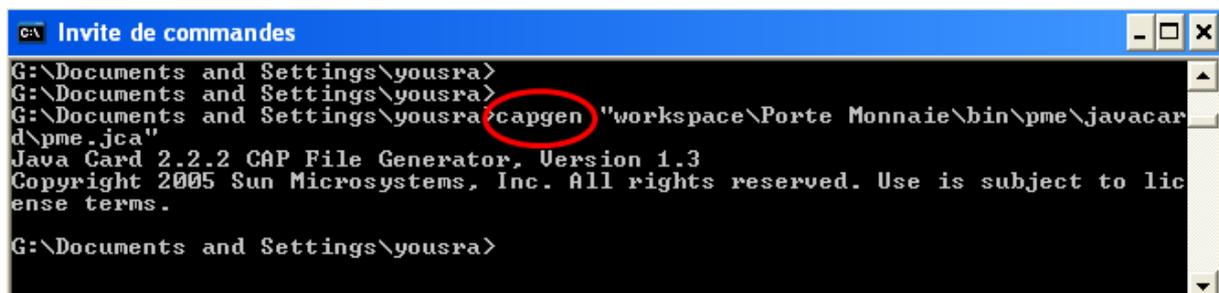
```

sconst_4;
bastore;
aload_0;
getfield_a_this 1; // reference pme/MonApplet.test
sconst_0;
baload;
putfield_s 0; // short pme/MonApplet.balance
aload_0;
new 6; // javacard/framework/OwnerPIN
dup;
sconst_3;
bspush 8;
invokespecial 4; // javacard/framework/OwnerPIN.<init>(BB)V
putfield_a 2; // reference pme/MonApplet.pin
aload_0;

```

Figure 6.5. Contenu du fichier **pme.jca**.

Nous allons, maintenant, convertir le fichier **pme.jca** en fichier CAP. La figure 6.6 illustre l'utilisation de l'outil **capgen**. Il produit en résultat le fichier **pme.cap**.



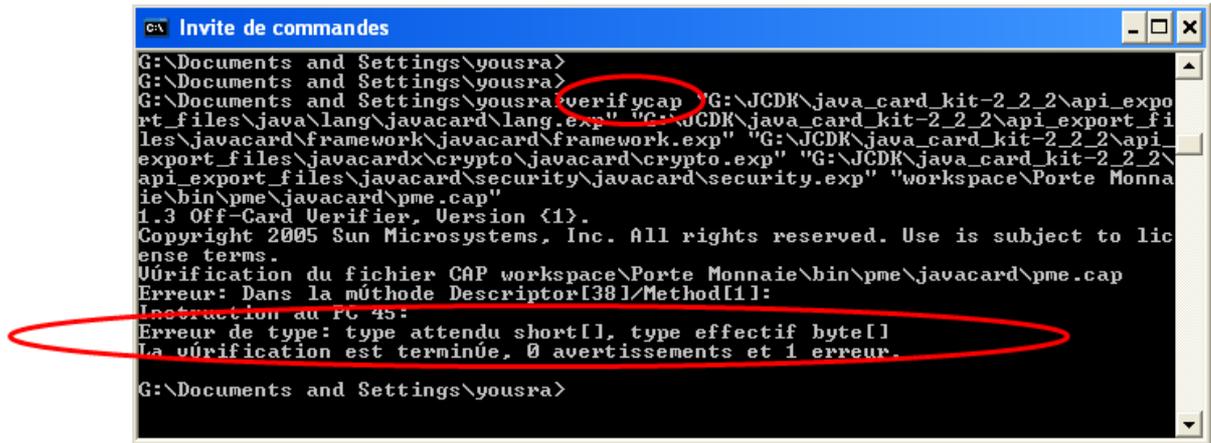
```

C:\ Invite de commandes
G:\Documents and Settings\yousra>
G:\Documents and Settings\yousra>
G:\Documents and Settings\yousra>capgen "workspace\Porte Monnaie\bin\pme\javacard\pme.jca"
Java Card 2.2.2 CAP File Generator, Version 1.3
Copyright 2005 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms.
G:\Documents and Settings\yousra>

```

Figure 6.6. Utilisation de **capgen**.

Ensuite, nous allons vérifier le contenu du fichier **pme.cap** à travers l'outil **verifycap**. Le vérifieur affiche une erreur de confusion de type, le résultat de la conversion est montré dans la figure 6.7.



```

G:\Documents and Settings\yousra>
G:\Documents and Settings\yousra>
G:\Documents and Settings\yousra>verifycap "G:\JCDK\java_card_kit-2_2_2\api_export_files\java\lang\javacard\lang.exp" "G:\JCDK\java_card_kit-2_2_2\api_export_files\javacard\framework\javacard\framework.exp" "G:\JCDK\java_card_kit-2_2_2\api_export_files\javacard\crypto\javacard\crypto.exp" "G:\JCDK\java_card_kit-2_2_2\api_export_files\javacard\security\javacard\security.exp" "workspace\Porte Monnaie\bin\pme\javacard\pme.cap"
1.3 Off-Card Verifier, Version {1}.
Copyright 2005 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms.
Vérification du fichier CAP workspace\Porte Monnaie\bin\pme\javacard\pme.cap
Erreur: Dans la méthode Descriptor[38]/Method[1]:
Instruction au PC 45:
Erreur de type: type attendu short[], type effectif byte[]
La vérification est terminée, 0 avertissements et 1 erreur.
G:\Documents and Settings\yousra>

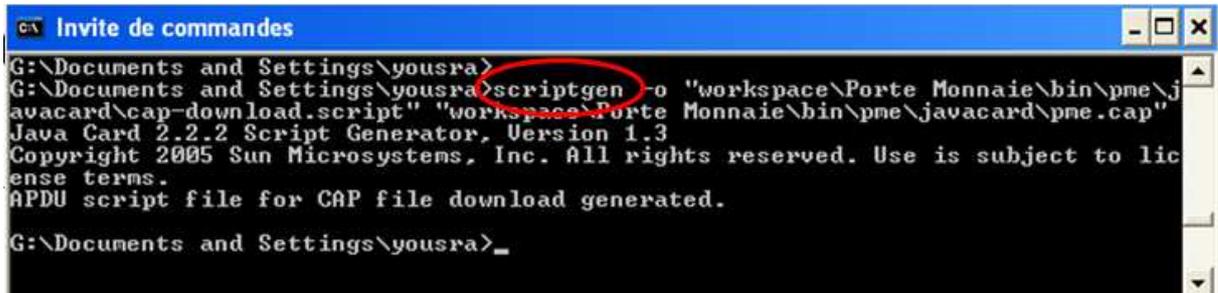
```

Figure 6.7. Détection de l'erreur par le vérifieur.

2.4.4. Installation du fichier CAP

Dans cette étape, nous allons charger le fichier **pme.cap** dans le simulateur du JCRE cref et créer une instance de l'applet **MonApplet**. Pour faire cette installation, nous devons suivre les étapes suivantes :

1. Utiliser le programme d'installation hors-carte scriptgen pour générer un fichier script à partir du fichier CAP. L'utilisation de l'outil scriptgen est illustrée dans la figure 6.8.



```

G:\Documents and Settings\yousra>
G:\Documents and Settings\yousra>scriptgen -o "workspace\Porte Monnaie\bin\pme\javacard\cap-download.script" "workspace\Porte Monnaie\bin\pme\javacard\pme.cap"
Java Card 2.2.2 Script Generator, Version 1.3
Copyright 2005 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms.
APDU script file for CAP file download generated.
G:\Documents and Settings\yousra>_

```

Figure 6.8. Utilisation de l'outil scriptgen.

En résultat, l'outil scriptgen crée le fichier de script **cap-download.script**.

2. Insérer au début du fichier script les commandes APDUs, de la mise sous tension de la carte et la sélection de l'installateur sur carte, suivantes :

```

powerup;

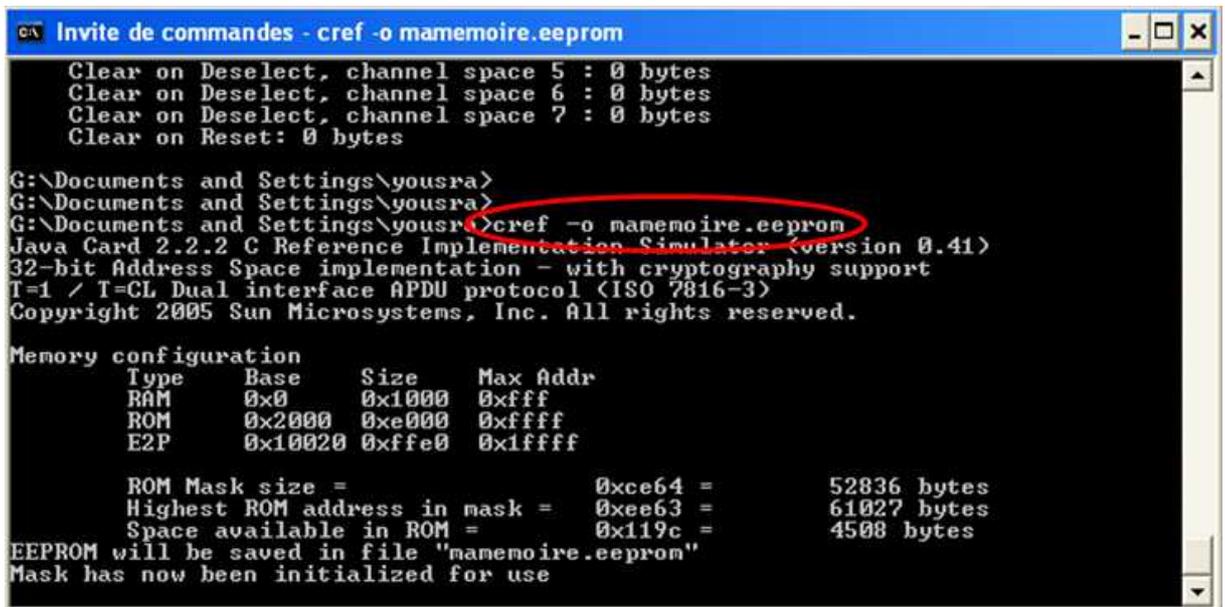
// Select the installer applet
0x00 0xA4 0x04 0x00 0x09 0xA0 0x00 0x00 0x00 0x62 0x03
0x01 0x08 0x01 0x7F;

```

3. Insérer en fin du fichier script les commandes APDUs, de la création de l'instance de l'applet et la mise hors tension de la carte, suivantes :

```
// create MonApplet applet
0x80 0xB8 0x00 0x00 0xd 0xb 0x01 0x02 0x03 0x04 0x05 0x06
0x07 0x08 0x09 0x00 0x00 0x00 0x7F;
powerdown;
```

4. Invoquer l'outil cref en lui donnant le nom du fichier qui va servir à sauvegarder l'image de l'EEPROM comme paramètre de sortie. L'utilisation de l'outil cref est illustrée dans la figure 6.9. Nous remarquons, dans la figure, l'affichage des informations concernant la carte simulée (version, tailles des différentes mémoires, etc.).



```

C:\> cref -o mamemoire.eeprom

Clear on Deselect, channel space 5 : 0 bytes
Clear on Deselect, channel space 6 : 0 bytes
Clear on Deselect, channel space 7 : 0 bytes
Clear on Reset: 0 bytes

G:\Documents and Settings\yousra>
G:\Documents and Settings\yousra>
G:\Documents and Settings\yousra> cref -o mamemoire.eeprom
Java Card 2.2.2 C Reference Implementation Simulator (version 0.41)
32-bit Address Space implementation - with cryptography support
T=1 / T=CL Dual interface APDU protocol (ISO 7816-3)
Copyright 2005 Sun Microsystems, Inc. All rights reserved.

Memory configuration
  Type      Base      Size      Max Addr
  RAM       0x0       0x1000    0xfff
  ROM       0x2000    0xe000    0xffff
  E2P       0x10020   0xffe0    0x1ffff

  ROM Mask size =                0xce64 =        52836 bytes
  Highest ROM address in mask =   0xee63 =        61027 bytes
  Space available in ROM =        0x119c =         4508 bytes
EEPROM will be saved in file "mamemoire.eeprom"
Mask has now been initialized for use

```

Figure 6.9. Utilisation de l'outil cref.

5. Invoquer le simulateur de l'interface de communication entre la carte et le CAD apdutool, dans une autre fenêtre de lignes de commandes, en lui donnant le nom du fichier **cap-download.script** comme paramètre. L'apdutool coopère avec l'installateur sur carte pour charger le contenu du fichier script dans l'image de l'EEPROM utilisée par le cref et créer une instance de l'applet. L'utilisation de l'outil apdutool est illustrée dans la figure 6.10.

```

G:\Documents and Settings\yourname\apdu tool "workspace\Porte Monnaie\bin\pme\javac
ard\cap-download.script"
Java Card 2.2.2 APDU Tool, Version 1.3
Copyright 2005 Sun Microsystems, Inc. All rights reserved. Use is subject to lic
ense terms.
Opening connection to localhost on port 9025.
Connected.
Received ATR = 0x3b 0xf0 0x11 0x00 0xff 0x01
CLA: 00, INS: a4, P1: 04, P2: 00, Lc: 09, a0, 00, 00, 00, 62, 03, 01, 08, 01, Le
: 00, SW1: 90, SW2: 00
CLA: 80, INS: b0, P1: 00, P2: 00, Lc: 00, Le: 00, SW1: 90, SW2: 00
CLA: 80, INS: b2, P1: 01, P2: 00, Lc: 00, Le: 00, SW1: 90, SW2: 00
CLA: 80, INS: b4, P1: 01, P2: 00, Lc: 17, 01, 00, 14, de, ca, ff, ed, 01, 02, 04
, 00, 01, 0a, 01, 02, 03, 04, 05, 06, 07, 08, 09, 00, Le: 00, SW1: 90, SW2: 00
CLA: 80, INS: bc, P1: 01, P2: 00, Lc: 00, Le: 00, SW1: 90, SW2: 00
CLA: 80, INS: b2, P1: 02, P2: 00, Lc: 00, Le: 00, SW1: 90, SW2: 00
CLA: 80, INS: b4, P1: 02, P2: 00, Lc: 20, 02, 00, 1f, 00, 14, 00, 1f, 00, 0f, 00
, 15, 00, 6e, 00, 18, 01, f4, 00, 0a, 00, 4f, 00, 00, 00, ed, 00, 00, 00, 00, 00
, 00, 02, Le: 00, SW1: 90, SW2: 00
CLA: 80, INS: b4, P1: 02, P2: 00, Lc: 02, 01, 00, Le: 00, SW1: 90, SW2: 00
CLA: 80, INS: bc, P1: 02, P2: 00, Lc: 00, Le: 00, SW1: 90, SW2: 00
CLA: 80, INS: b2, P1: 04, P2: 00, Lc: 00, Le: 00, SW1: 90, SW2: 00
CLA: 80, INS: b4, P1: 04, P2: 00, Lc: 18, 04, 00, 15, 02, 03, 01, 07, a0, 00, 00
, 00, 62, 01, 01, 00, 01, 07, a0, 00, 00, 00, 62, 00, 01, Le: 00, SW1: 90, SW2:
00
CLA: 80, INS: bc, P1: 04, P2: 00, Lc: 00, Le: 00, SW1: 90, SW2: 00

```

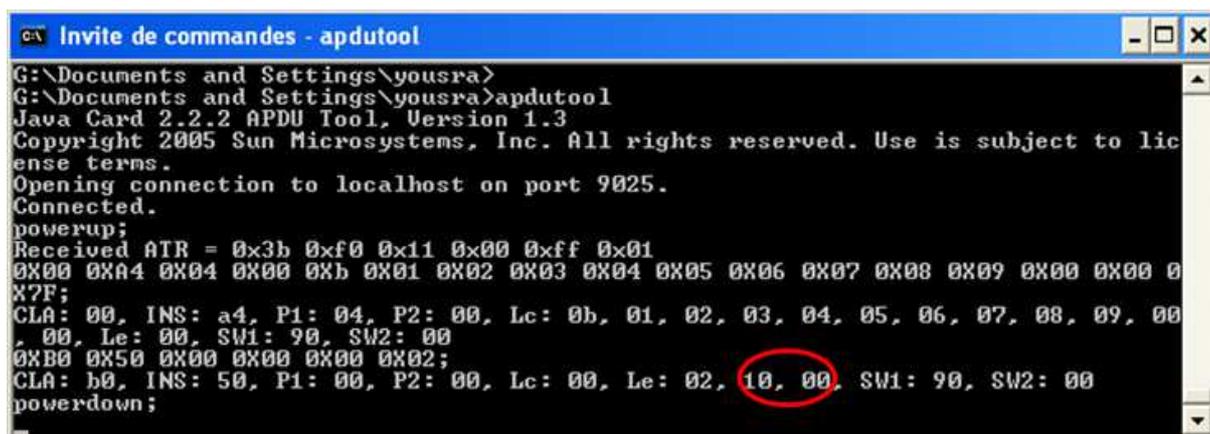
Figure 6.10. Utilisation de l'outil apdu tool.

Pour chaque commande APDU contenue dans le fichier script, le cref retourne une réponse APDU qui contient un Statut Word. Si le Statut Word est égale à 90 00 sa veut dire que la commande a été exécutée normalement. Sinon, il y a eu des erreurs dans l'exécution de la commande.

2.4.5. Exécution de l'applet

Jusqu'ici, on a créé une instance de l'applet **MonApplet** et on la sauvegardé dans le fichier **mamemoire.eeprom**. Pour exécuter cette applet, nous allons suivre les étapes suivantes :

1. Invoquer le cref en lui donnant comme paramètre d'entrée et de sortie le fichier **mamemoire.eeprom**.
2. Invoquer, dans une autre fenêtre de lignes de commandes, l'apdu tool (sans paramètres). L'apdu tool va connecter au simulateur de la carte cref et va nous donner la main pour écrire des commandes APDUs.
3. Nous allons entrer la commande de la sélection de l'applet et ensuite la commande qui contient l'instruction de consultation du solde. La figure 6.11 illustre l'exécution de l'applet tout en montrant le résultat affiché (la valeur du solde qui est 10,00).



```

G:\Documents and Settings\yousra>
G:\Documents and Settings\yousra>apdutool
Java Card 2.2.2 APDU Tool, Version 1.3
Copyright 2005 Sun Microsystems, Inc. All rights reserved. Use is subject to lic
ense terms.
Opening connection to localhost on port 9025.
Connected.
powerup;
Received ATR = 0x3b 0xf0 0x11 0x00 0xff 0x01
0X00 0XA4 0X04 0X00 0Xb 0X01 0X02 0X03 0X04 0X05 0X06 0X07 0X08 0X09 0X00 0X00 0
X7F;
CLA: 00, INS: a4, P1: 04, P2: 00, Lc: 0b, 01, 02, 03, 04, 05, 06, 07, 08, 09, 00
, 00, Le: 00, SW1: 90, SW2: 00
0XB0 0X50 0X00 0X00 0X00 0X02;
CLA: b0, INS: 50, P1: 00, P2: 00, Lc: 00, Le: 02, 10, 00, SW1: 90, SW2: 00
powerdown;

```

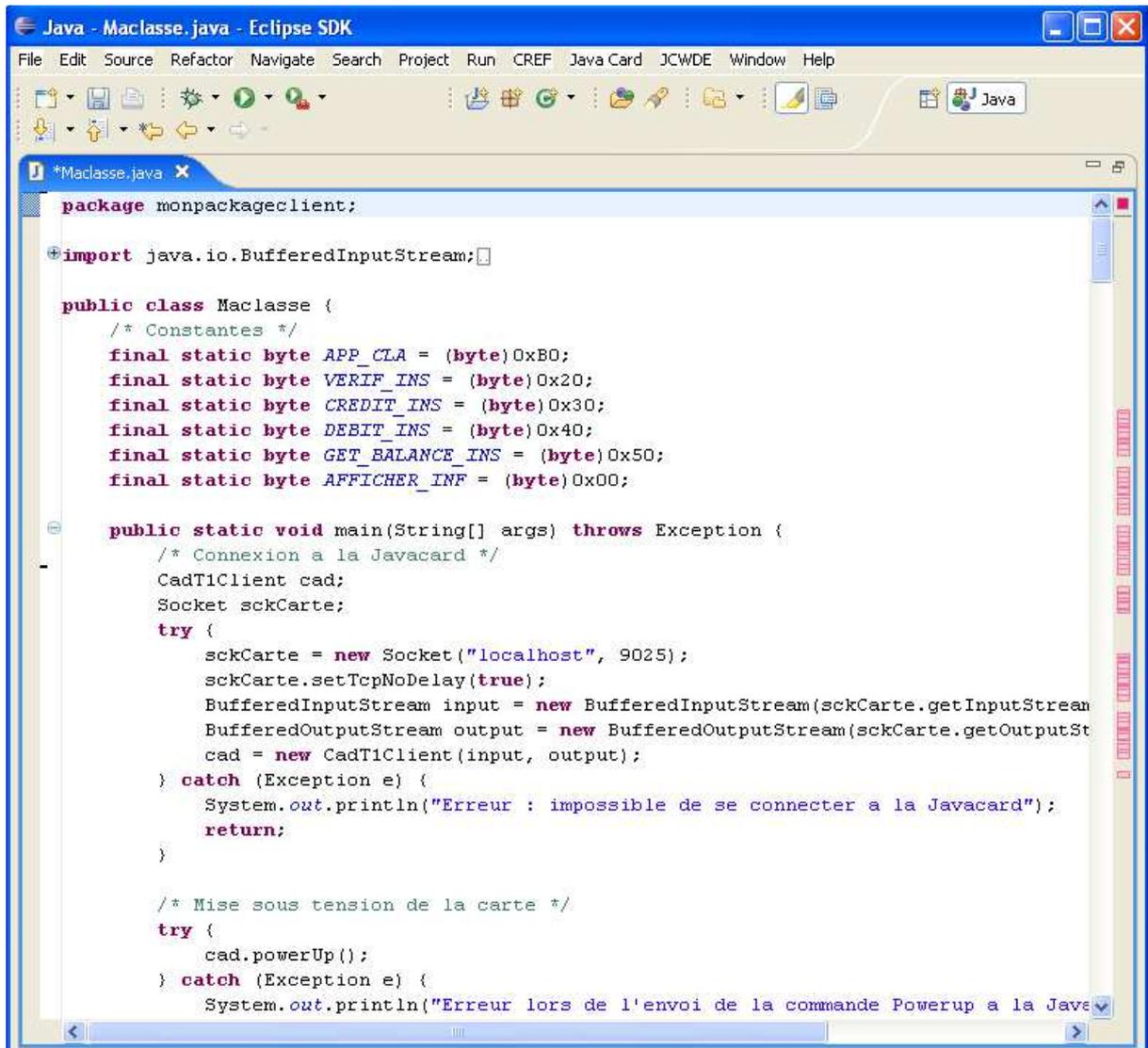
Figure 6.11. Exécution de l'applet MonApplet.

2.4.6. Utilisation d'une application cliente

L'applet installée sur la Java Card peut collaborer avec une application cliente qui se situe dans l'ordinateur qui communique avec la carte à travers le CAD. L'application cliente offre à l'utilisateur une interface simple d'utilisation de la carte (exécution des applets). Elle dissimule les différentes opérations de connexion à la carte, sélection des applets et échanges des commandes APDUs.

L'application cliente peut être un programme Java développé et exécuté dans un environnement de développement, ici nous utilisons l'environnement Eclipse. Nous devons ajouter la bibliothèque de Java Card **apduio.jar** à l'application pour qu'elle puisse communiquer avec le simulateur de l'interface de communication apdutool.

L'application doit contenir les instructions de connexion à l'apdutool, de sélection des applets, et d'envois et de lecture de commandes APDUs. La figure 6.12 montre un échantillon du code de l'application cliente que nous avons développé.



```
Java - Maclasse.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run CREF JavaCard JCWDE Window Help

package monpackageclient;

import java.io.BufferedReader;

public class Maclasse {
    /* Constantes */
    final static byte APP_CLA = (byte)0xB0;
    final static byte VERIF_INS = (byte)0x20;
    final static byte CREDIT_INS = (byte)0x30;
    final static byte DEBIT_INS = (byte)0x40;
    final static byte GET_BALANCE_INS = (byte)0x50;
    final static byte AFFICHER_INF = (byte)0x00;

    public static void main(String[] args) throws Exception {
        /* Connexion a la Javacard */
        CadT1Client cad;
        Socket sckCarte;
        try {
            sckCarte = new Socket("localhost", 9025);
            sckCarte.setTcpNoDelay(true);
            BufferedInputStream input = new BufferedInputStream(sckCarte.getInputStream());
            BufferedOutputStream output = new BufferedOutputStream(sckCarte.getOutputStream());
            cad = new CadT1Client(input, output);
        } catch (Exception e) {
            System.out.println("Erreur : impossible de se connecter a la Javacard");
            return;
        }

        /* Mise sous tension de la carte */
        try {
            cad.powerUp();
        } catch (Exception e) {
            System.out.println("Erreur lors de l'envoi de la commande Powerup a la Javacard");
        }
    }
}
```

Figure 6.12. Code de l'application **Application client**.

Pour pouvoir exécuter l'application, nous allons suivre les mêmes étapes précédentes jusqu'au chargement et la création d'instance de l'applet dans le fichier image de la mémoire. Ensuite, nous allons lancer le cref, en ligne de commandes, à partir de l'image mémoire créée et exécuter notre application, comme une application Java ordinaire, dans Eclipse. L'exécution est illustrée dans la figure 6.13.

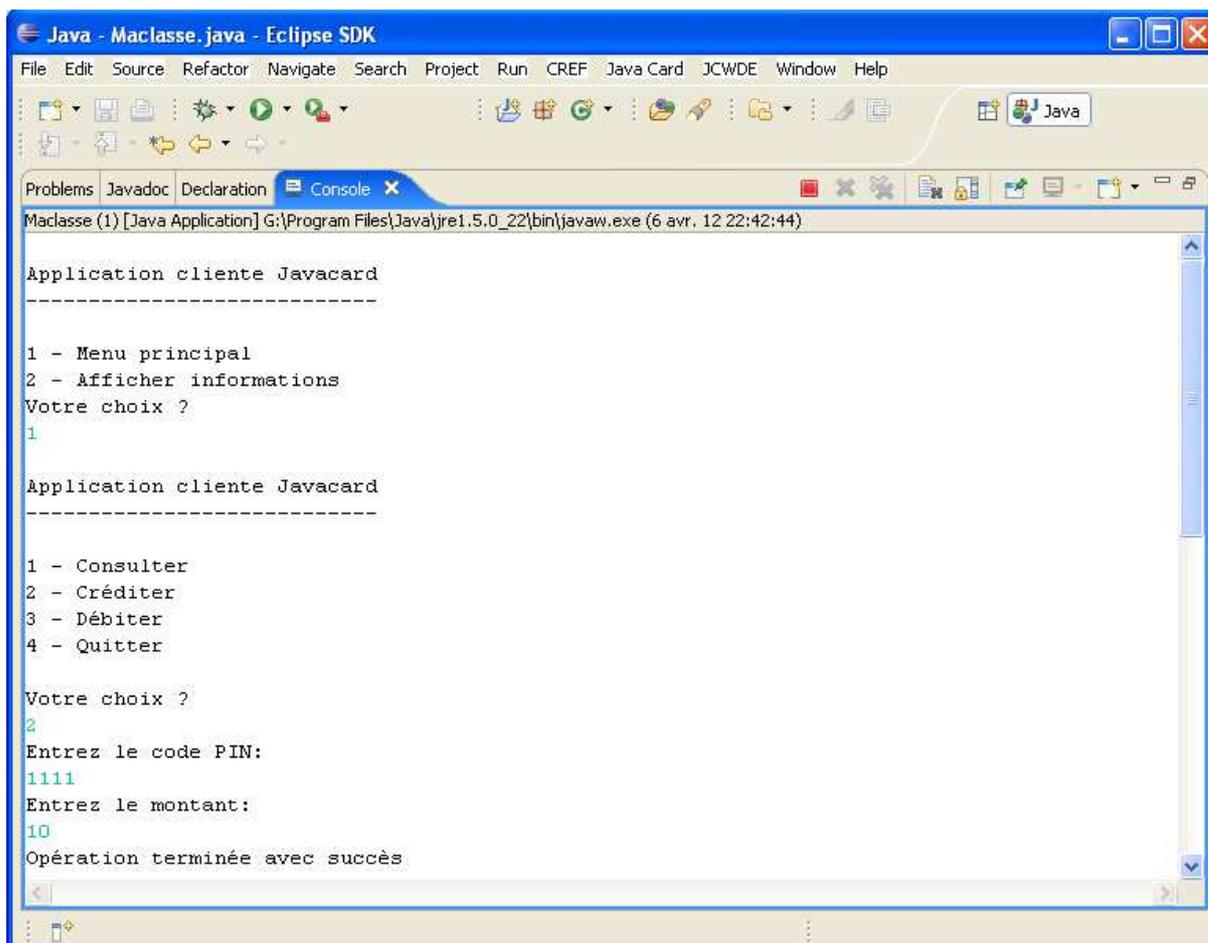


Figure 6.13. Utilisation de l'application cliente.

Dans ce qui suit, nous allons utiliser le kit de développement (Javacard Development Kit 2.2.2) présenté, ici, pour exécuter les différentes applications que nous allons développer, démontrer l'exploitation des attaques dans ses applications, et montrer les résultats de la vérification.

3. ATTAQUE D'ACCES A LA MEMOIRE COMPLETE SUR UNE JAVA JARD

Dans cette section, nous allons présenter une attaque qui permet l'accès (en lecture et écriture) à la mémoire persistante complète d'une Java Card. Ensuite, nous allons décrire l'utilisation de cette attaque dans notre mémoire.

3.1. Description de l'attaque

Nous présentons ici, une attaque concrète sur un type particulier de Java Card [52] [53], issue des travaux de [6]. Cette attaque exploite les vulnérabilités de l'implémentation de la machine virtuelle Java Card afin d'entraîner la lecture ou l'écriture dans presque tous les emplacements de la mémoire persistante de la carte.

L'attaque utilise la technique connue de confusion type de la machine virtuelle Java Card en exploitant une vulnérabilité présente dans le mécanisme de transaction offert par la plateforme Java Card. Les attaques de confusion de type permettent d'accéder aux données privées d'une application et de lire et d'écrire dans des emplacements de mémoire arbitraires sur la carte. La conséquence est la possibilité de lire et de modifier le code et les données d'autres applets présentes sur la carte.

Notons ici que cette attaque, bien que très puissante dans sa nature (globale, c'est à dire donnant accès à toute la carte) est seulement nocive pour les cartes ouvertes. Nous entendons par là que seules les cartes qui ont la possibilité d'installer de nouvelles applets sont sensibles à ce genre d'attaque.

3.1.1. Travaux connexes

Les attaques logiques sur les Java Cards sont maintenant bien décrites dans la littérature. Dans [35] Marc Witteman donne les principes généraux de ces attaques et donne quelques exemples de la manipulation de bytecode avant son installation sur la Java Card, ainsi il fournit l'idée d'utiliser le mécanisme de pare-feu pour obtenir des résultats similaires à la manipulation de bytecode, à savoir l'introduction de la confusion de type dans la JCVM.

Basés sur ces idées, les développeurs de l'attaque que nous allons décrire, ont essayé différentes techniques pour créer de la confusion de type sur plusieurs Java Cards concrètes et ont décrit les résultats et le comportement des mécanismes de protection mis en place sur ces cartes.

Ils ont réussi à trouver une attaque qui donne des résultats similaires à celle que nous allons présenter, basée sur l'exploitation de l'instruction de bytecode `getstatic` et la manipulation de bytecode avant son installation sur la carte. Comme l'attaque de `getstatic` repose en partie sur la manipulation de bytecode avant son installation, elle ne passera pas facilement la vérification de bytecode. Les auteurs de [54] ont réclamé des solutions de contournement possibles pour passer le vérifieur, toutefois, cela rend l'attaque encore plus complexe. Notre attaque, d'autre part, est très simple et ne nécessite aucune manipulation directe de bytecode. [6]

3.1.2. Confusion de type

Créer de la confusion de type dans la machine virtuelle Java qui s'exécute sur la carte, signifie de tromper la JVM à créer deux références de deux différents types pointant vers le même emplacement mémoire. Dans notre attaque, le but est d'obtenir une référence à un tableau de type byte et une référence à un tableau de type short au même bloc de mémoire. En supposant que le bloc de mémoire a été à l'origine (légalement) alloué en tant que tableau byte, y accéder comme un tableau short donne la possibilité de lire les données au-delà des limites légales du tableau byte. Limitée par la taille du tableau, la lecture d'un nombre donné d'éléments du tableau short permet d'accéder aux blocs de mémoire deux fois la taille du tableau byte d'origine. La seconde moitié du tableau short est la zone mémoire normalement inaccessible au-delà du tableau byte. La figure 6.14 illustre ceci. [6]

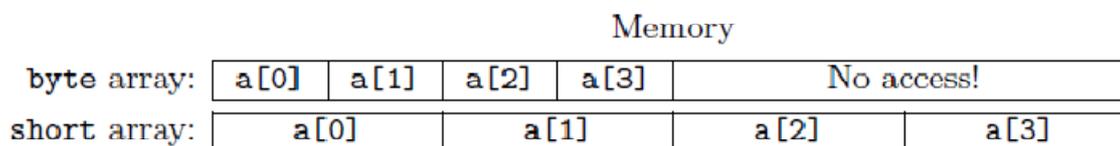


Figure 6.14. La confusion de type dans les tableaux. [6]

Lorsque la taille du tableau byte d'origine est assez grande, l'accès aux blocs de mémoire au-delà du tableau nous donnera l'accès à d'autres données sur la carte. En général, ces données pourraient être n'importe quoi, en fonction de l'organisation mémoire de la carte. Par exemple, nous pouvons atteindre la zone de mémoire réservée pour une autre applet, ou la zone de mémoire où le système d'exploitation ou la JVM conservent leurs données associés à une application donnée.

La façon la plus simple d'introduire une telle confusion de type dans une Java Card est d'installer sur la carte un bytecode mal typé (contient une confusion de type). La confusion de type expliquée ci-dessus ne nécessite qu'une mineure modification dans le bytecode produit par le compilateur Java. Cependant, le problème est qu'un tel code modifié et hostile ne sera pas accepté par le vérifieur de bytecode avant son chargement sur la carte : le vérifieur va détecter que le bytecode est mal typé et va refuser la signature de l'applet. Par conséquent on ne peut pas l'installer sur la carte qui va vérifier cette signature. Donc, nous avons besoin d'une autre façon d'introduire la confusion de type. Une vulnérabilité dans l'implémentation du mécanisme de transaction nous permet de le faire. [6]

3.1.3. La vulnérabilité dans le mécanisme de transaction

Le mécanisme de transaction de Java Card permet au programmeur de protéger la cohérence des données persistantes stockées sur la carte dans la mémoire EEPROM. Les champs principaux de la classe applet sont de notables exemples de données persistantes qui sont stockées dans l'EEPROM. L'autre type de mémoire qu'utilise Java Card est la RAM, elle est utilisée pour toutes les variables locales et les calculs temporaires. Le mécanisme de transaction n'est concerné qu'avec le contenu de la mémoire EEPROM, la mémoire RAM n'est pas affectée par les transactions. L'accès au mécanisme de transaction est fourni par les méthodes de l'API Java Card suivantes:

- `JCSystem.beginTransaction ()` : commence une transaction atomique.
- `JCSystem.commitTransaction ()` : commet une transaction atomique, toutes les modifications dans la mémoire persistante depuis `beginTransaction ()` sont exécutées dans une étape atomique.
- `JCSystem.abortTransaction ()` : annule la transaction, toutes les modifications dans la mémoire persistante depuis `beginTransaction ()` sont ignorées. Un avortement implicite peut aussi être causé par une terminaison d'exécution d'un programme anormale (par exemple une perte de puissance au retrait de la carte).

Dans Java Card, les objets alloués avec l'instruction `new` sont placés dans la mémoire EEPROM. Selon les règles de transaction pour la mémoire persistante, toute allocation d'objet possible à l'intérieur d'une transaction doit être annulée dans le cas d'avortement de la transaction. La spécification du langage Java Card stipule que, dans ce cas les références à tous les objets alloués dans le cadre d'une transaction doivent être remises à la valeur `null` et la mémoire allouée par ces objets peut être utilisée pour stocker d'autres données (libérée). C'est là où une possible vulnérabilité dans le mécanisme de transaction peut être exploitée. Dans le mécanisme de transaction, seules les références conservées dans des variables persistantes (champs de classe) sont remises à `null`. Les références conservés dans les variables locales et transitoires sont laissées intactes pendent une annulation de transaction. Ces références intactes contiennent des pointeurs vers des zones mémoire qui sont libérées par l'avortement de la transaction. Une ultérieure allocation de la mémoire va réutiliser cette référence et si cette nouvelle allocation a un différent type que celui de la référence avortée, une confusion de type est produite. L'extrait de code suivant illustre cette idée :

```
short[] arrayS; // instance field, persistent
byte[] arrayB; // instance field, persistent
...
short[] arraySlocal = null; // local variable, transient
JCSYSTEM.beginTransaction();
arrayS = new short[1]; // allocation to be rolled back
arraySlocal = arrayS; // save the reference in local variable
JCSYSTEM.abortTransaction(); // arrayS is reset to null,
                             // but not arraySlocal
arrayB = new byte[10]; // arrayB gets the same reference as
                       // arrayS used to have, this can be tested:
if((Object)arrayB == (Object)arraySlocal) ... // this
condition is true
```

A la fin d'exécution de ce morceau de code, nous aurons deux variables de différents types (un tableau de byte et un tableau de short) avec la même référence dans la mémoire.

Notons que cela est un code Java Card parfaitement légal et aucune modification du bytecode produit par le compilateur n'a été faite. Ainsi, ce programme parvient à passer le vérifieur de bytecode et peut être installé sur la carte. [6]

3.1.4. Réalisation de l'attaque

Nous avons maintenant toutes les techniques qu'on a besoin pour effectuer l'attaque. Notre but est de lire la zone de mémoire de l'applet, où les données sont conservées, en particulier les informations sur les tableaux. Puis en changeant ces données, nous pourrions essayer d'exploiter la carte.

Pour pouvoir lire les données de l'applet, il faut allouer un tableau de type byte assez large, et en utilisant la technique de confusion de type, lire le bloc de mémoire qui suit ce tableau. En effet, selon [6], cela a donné des résultats significatifs, à savoir les données lues contenaient des informations d'allocation de tableaux. L'étude expérimentale (répartition des différents tableaux) et l'analyse des données ont révélé comment les données contenant les informations sur les tableaux sont stockées dans la mémoire de l'applet. Pour chaque tableau alloué de type byte on trouve une séquence de 5 octets, comme celle-ci:

0B80FFA54C

Le premier octet indique le type d'éléments du tableau (0B définit un tableau de byte). Les deux octets suivants 80FF, à l'exclusion du premier bit, définissent la longueur du tableau, dans ce cas 255. Les deux derniers octets A54C contiennent le pointeur effectif aux données du tableau.

À ce stade, rien ne nous empêche de modifier ces données pour voir si elles ont vraiment un impact sur la variable d'instance correspondante au tableau de byte dans l'applet. En effet nous pourrions changer la taille du tableau sans aucun problème, ainsi que la valeur de son pointeur, et appeler le code Java pour lire la référence du tableau modifié. En passant par tout le rang des pointeurs possibles, nous pourrions, en théorie, lire (et écrire) dans des emplacements de mémoire arbitraires sur la carte. En pratique, on ne réussit pas à lire toutes les emplacements des pointeurs.

3.1.5. Exploitation de l'attaque

Dans cette section, nous discutons la structure générale de l'applet exploitant l'attaque précédente.

Durant la phase de la préparation de l'attaque, la confusion de type est introduite par les instructions présentées dans la section 3.1.3 (mécanisme de transaction). Après l'exécution de ces instructions, il y a deux variables persistantes dans l'applet appelées `arrayB` et `arrayS` qui pointent sur le même bloc de mémoire. Les types correspondants de ces variables sont tableau byte et tableau short.

Puis l'applet alloue un tableau spécial pour cibler notre attaque. Sa référence est stockée dans la variable `arrayMutable`. Plus tard, après que l'attaque est préparée (la confusion de type est appliquée et les données de l'applet peuvent être accédées), nous changerons ces données pour rendre la variable `arrayMutable` pointer sur des emplacements de mémoire arbitraires. Initialement la longueur de `arrayMutable` est fixée à une valeur prédéfinie, afin que nous puissions facilement trouver ses données correspondantes. Si nous fixons la longueur à 2, nous devons avoir la séquence d'octets 0B8002 dans les données de ce tableau. Une fois que les données correspondantes sont trouvées, leur emplacement est stocké en permanence dans l'applet dans une variable appelée `index`.

À ce stade, l'attaque est totalement préparée, l'applet est maintenant prête à lire des blocs de mémoire arbitraires. Cela se fait avec une commande APDU séparée qui contient la longueur et la valeur du pointeur qui vont être injectées dans la référence de `arrayMutable`. Après cela, le contenu du tableau, et donc le bloc mémoire demandé au pointeur donné, peuvent être lus et envoyés à l'hôte.

3.1.6. Résultats de l'attaque

Nous présentons, ici, les résultats des expérimentations effectuées et présentées par les auteurs de [6].

À ce stade, c'est à l'application cliente d'appeler l'applet avec toutes les valeurs de pointeurs nécessaires pour lire la mémoire de la carte bloc par bloc. Les auteurs de [6] ont écrit une petite application cliente pour juste faire cela. Son exécution sur la carte de test a permis de lire environ 48Ko de données de blocs continus de différentes tailles à partir de différents endroits.

Les adresses dans la plage de 0000 à FFFF n'ont pas été toutes lisibles. Ce devait être prévu.

La spécification de la carte déclare la taille de l'EEPROM à 32KB. Alors qu'ils ont réussi à lire plus que ça, ils ont supposé que des parties de RAM de la carte et / ou de la ROM sont aussi accédées.

3.2. Réalisation de l'attaque sur l'applet « MonApplet »

Dans cette section, nous allons utiliser le kit de développement pour la plate-forme Java Card 2.2.2 (Javacard Development Kit 2.2.2) pour simuler une carte de crédit et nous essayons d'exploiter l'attaque d'accès à la mémoire complète sur une application de porte monnaie électronique contenue dans cette carte, dans le but de modifier dans le code de cette applet. La modification dans le code de l'applet va permettre au porteur de la carte de débiter des sommes d'argent qui dépassent son solde et donc d'obtenir un gain financier.

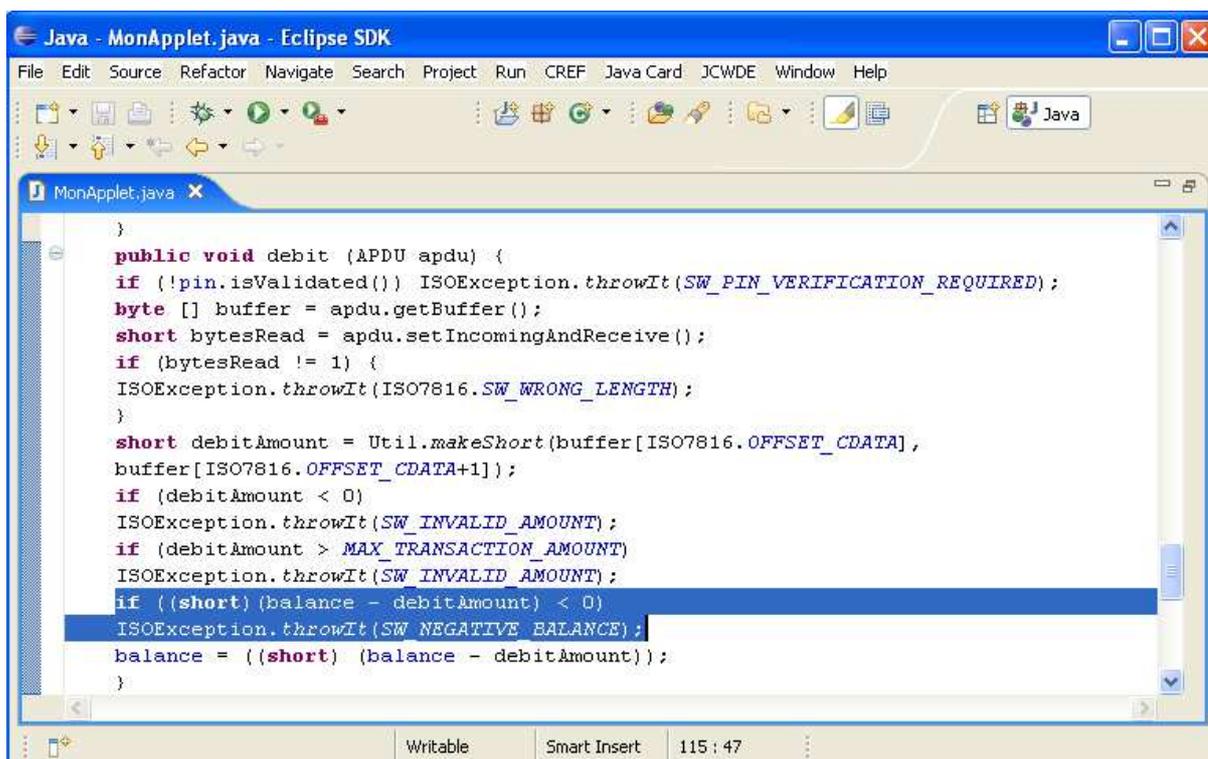
3.2.1. Cible d'attaque

La cible de notre attaque est la modification dans le code de l'applet **MonApplet** contenue dans le package **pme** présentée dans la section 2.4.

L'applet fournit une fonction de débit. Cette fonction permet à l'utilisateur de la carte de débiter une somme d'argent de son compte. Elle fonctionne comme suit :

- L'utilisateur entre le code PIN de la carte.
- Si le code PIN est valide, on lui permet d'entrer la somme d'argent qu'il voudrait débiter.
- La fonction de débit (dans l'applet) vérifie, entre autres choses, que le solde de l'utilisateur est suffisant pour effectuer ce débit.
- Si le solde est suffisant, l'utilisateur peut débiter de son compte.
- Sinon on relève une exception, on annule l'opération de débit, et on affiche un message d'erreur à l'utilisateur.

Nous voulons, en utilisant l'attaque de lecture de mémoire complète présentée précédemment, accéder au code de cette applet après son installation sur la carte et modifier dans ce code. Précisément, effacer les instructions qui permettent la vérification de la suffisance du solde pour effectuer le débit et l'exception engendrée dans ce cas. La figure 6.15 montre le code de la fonction de débit et les instructions que nous voudrions effacer (les instructions sélectionnées).



```
Java - MonApplet.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run CREF Java Card JCWDE Window Help

MonApplet.java x
}
public void debit (APDU apdu) {
    if (!pin.isValidated()) ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);
    byte [] buffer = apdu.getBuffer();
    short bytesRead = apdu.setIncomingAndReceive();
    if (bytesRead != 1) {
        ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
    }
    short debitAmount = Util.makeShort(buffer[ISO7816.OFFSET_CDATA],
        buffer[ISO7816.OFFSET_CDATA+1]);
    if (debitAmount < 0)
        ISOException.throwIt(SW_INVALID_AMOUNT);
    if (debitAmount > MAX_TRANSACTION_AMOUNT)
        ISOException.throwIt(SW_INVALID_AMOUNT);
    if ((short) (balance - debitAmount) < 0)
        ISOException.throwIt(SW_NEGATIVE_BALANCE);
    balance = ((short) (balance - debitAmount));
}
Writable Smart Insert 115 : 47
```

Figure 6.15. Code de la fonction de débit.

3.2.2. Réalisation de l'attaque

Nous avons expliqué précédemment, que pour pouvoir réaliser ce type d'attaques nous avons besoin d'une carte à puce ouverte (permet le chargement de nouvelles applications), ce qui est le cas dans notre simulateur. Nous allons commencer, ici, par installer le package **pme** de l'applet ciblée sur le simulateur de la carte. Et ensuite, nous allons installer une autre applet sur cette carte qui contient le code malicieux qui réalise l'attaque, présenté précédemment.

Nous avons écrit une applet appelée **MonAttack**, contenue dans un package appelé **packageAttack**, qui permet d'ajouter une nouvelle fonctionnalité à la carte : consulter des informations sur la carte (titulaire de la carte, date d'activation, date d'expérimentation, etc.). La figure 6.16 montre le résultat de l'exécution de cette nouvelle applet.

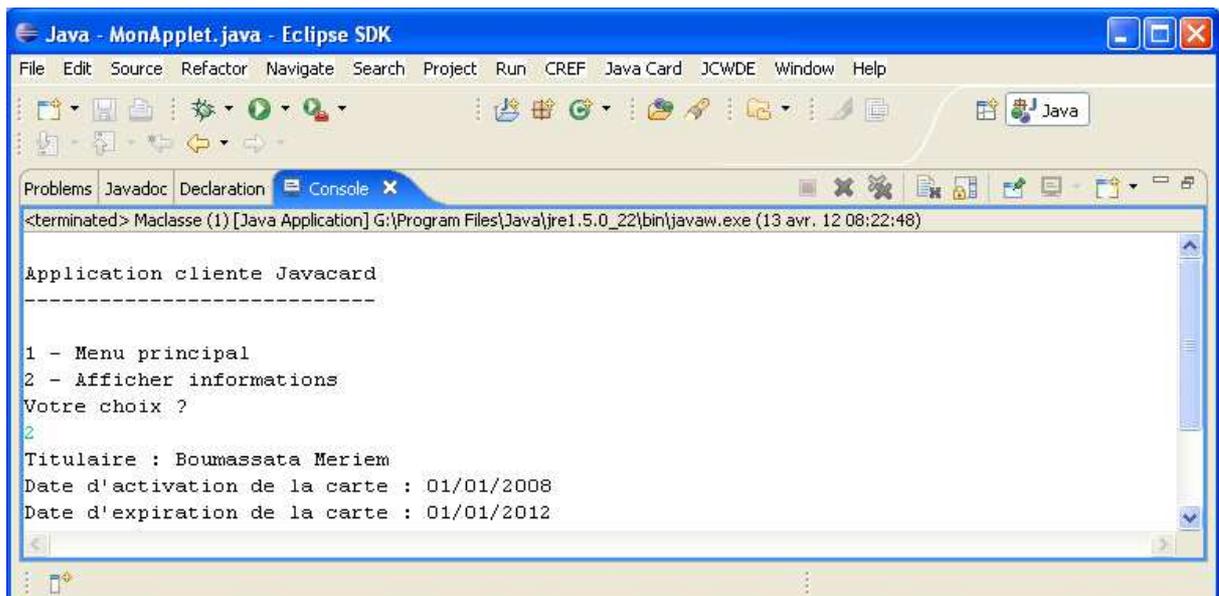


Figure 6.16. Exécution de **MonAttack** avant l'ajout de l'attaque.

Cette applet ne contient pas encore le code de l'attaque. Nous allons ajouter, maintenant, le code décrit dans la section 3.1.5 à cette applet et adapter ce code pour pouvoir réaliser l'attaque dans notre projet. Et en fin, nous allons réinstaller le package **packageAttack** dans le simulateur de la carte pour réaliser l'attaque.

Notons, que la vérification de fichier CAP **packageAttack.cap** contenant le code de l'attaque, avant son installation sur la carte, par le vérifieur `verifycap` indique que le fichier ne contient pas d'erreurs et donc qu'il peut être installé sur la carte normalement. La figure 6.17 montre le résultat de la vérification.

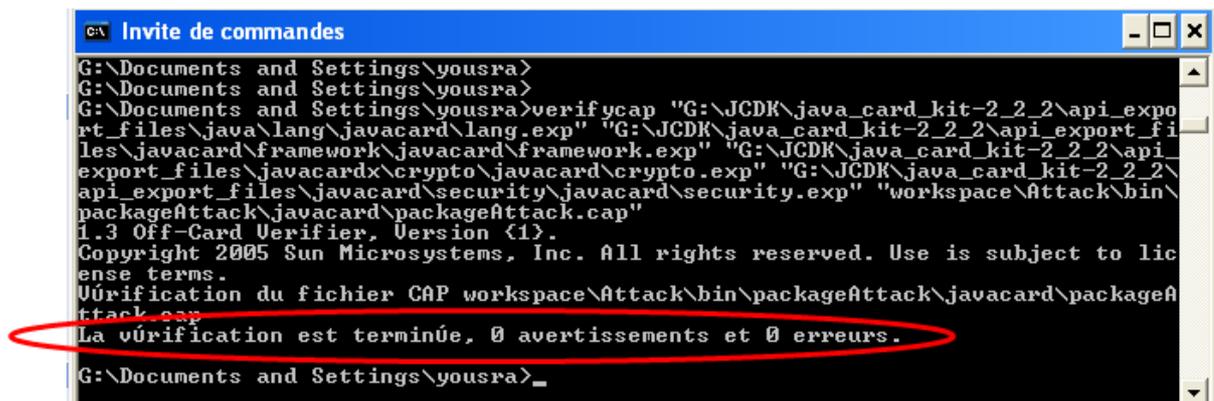


Figure 6.17. Vérification du fichier **packageAttack.cap**.

3.2.3. Résultats de l'attaque

L'efficacité de l'attaque de lecture de mémoire persistante complète d'une Java Card est prouvée dans les travaux de [6]. Nous avons utilisé cette attaque pour réaliser un gain financier en l'exploitant sur une application de porte monnaie électronique présente sur une carte de crédit de type Java Card.

Pour voir le résultat de cette attaque, nous avons utilisé l'environnement de développement pour la plate-forme Java Card 2.2.2 (Javacard Development Kit 2.2.2) qui contient les différents outils nécessaires pour simuler le chargement et l'exécution d'applets sur une Java Card.

Les différents outils de ce kit, à l'aide de l'environnement de développement Eclipse, nous ont permis de développer les applets **MonApplet** et **MonAttack**, d'effectuer les transformations nécessaires pour ces applets, de vérifier le code de ces applets et de les charger et installer sur le simulateur de la carte en créant une image mémoire de la carte, et en fin d'exécuter ces applets en coopérant avec une application cliente.

Cependant, nous avons rencontré des problèmes lors de l'exécution de l'applet **MonAttack** après l'ajout du code de l'attaque à cette applet.

Nous avons déjà expliqué que la réalisation de cette attaque repose sur l'organisation des données présentes sur la mémoire persistante de la carte. Le problème avec le simulateur c'est qu'il ne possède pas une vraie mémoire EEPROM, mais il utilise une image mémoire contenue dans un fichier texte enregistré dans le disque. Cette image mémoire fonctionne parfaitement pour l'exécution d'instructions d'une applet, à l'exception des instructions qui touchent directement aux blocs de mémoire EEPROM. Dans notre cas l'instruction d'avortement `JCSystem.abortTransaction ()` qui permet d'annuler toutes les modifications dans la mémoire persistante depuis l'instruction `JCSystem.beginTransaction ()`.

Le résultat de l'exécution de cette instruction dans notre application, est le relèvement d'une exception, la déconnection de la carte, et l'affichage d'un message d'erreur. Ceci est illustré dans la figure 6.18.

Le but dans notre projet n'est pas de prouver l'efficacité de l'attaque présentée ici, mais de prouver à travers le code contenant cette attaque l'efficacité du nouveau vérifieur hors-carte que nous avons réalisé.

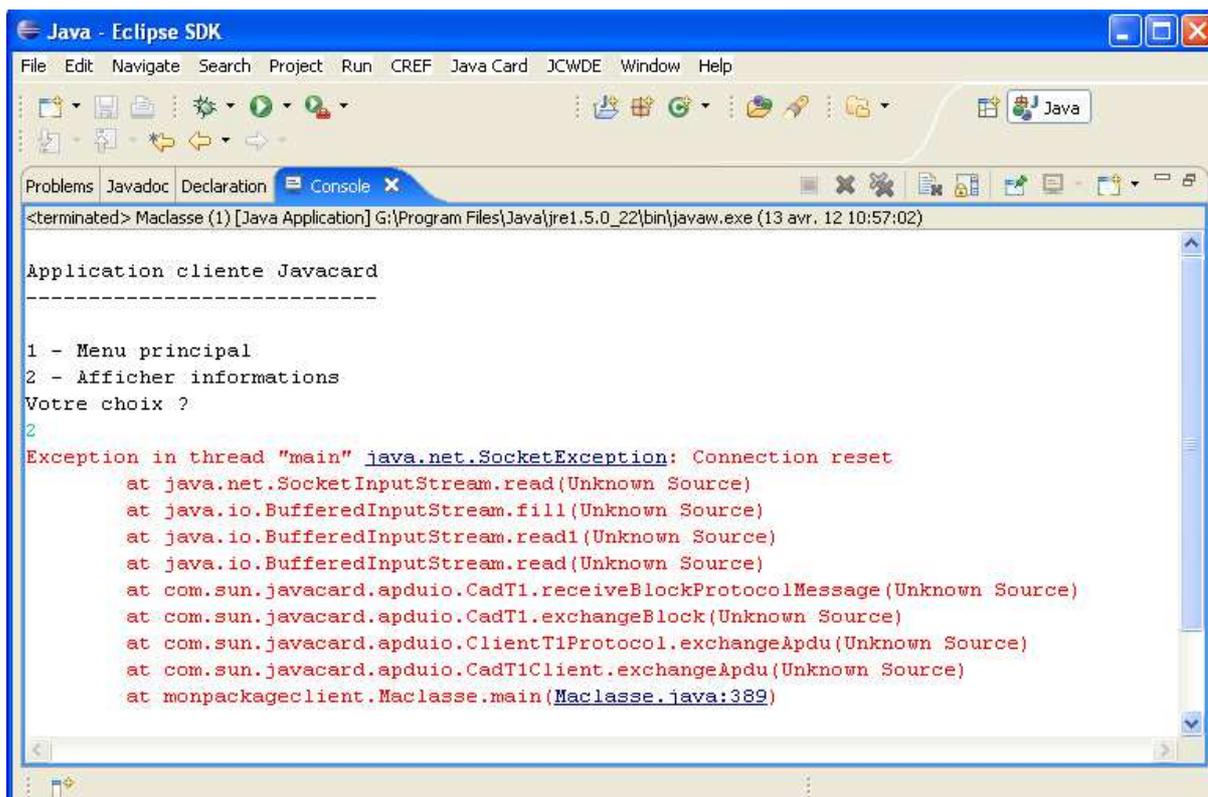


Figure 6.18. L'exception causée par `JCSystem.abortTransaction()`.

4. CONTRIBUTION

Nous allons présenter, dans cette section, le travail que nous avons réalisé au sein de ce mémoire, qui consiste à introduire un nouveau module de vérification dans le vérifieur hors-carte de Java Card, en utilisant l'environnement de développement pour la plate-forme Java Card 2.2.2 (Javacard Development Kit 2.2.2). Ce nouveau module de vérification nous permettra de détecter la présence d'une confusion de type causée par l'attaque présentée dans la section précédente.

Notons, ici, que l'attaque d'accès à la mémoire complète qu'on a utilisé pour réaliser l'attaque sur l'application de porte monnaie électronique, est un exemple de l'exploitation de la confusion de type causée par l'utilisation de la vulnérabilité présente dans le mécanisme de transaction. Cette confusion de type peut être exploitée de différentes manières, causant différents types d'attaques qui peuvent être exploitées sur différentes cartes en fonction de leurs caractéristiques et de l'organisation interne de leurs mémoires.

Le nouveau vérifieur que nous avons produit, au sein de ce travail, permet de vérifier la présence de la cause de la confusion de type elle-même, qui est l'exploitation de la vulnérabilité du mécanisme de transaction, et par conséquent, détecter n'importe quelle attaque exploitant cette confusion de type.

4.1. Cause de l'attaque

Java Card a introduit le mécanisme de transaction (présenté dans le Chapitre 4) dans le but de garantir l'atomicité des opérations qui peuvent être interrompues par des circonstances externes (coupure de courant, retrait de la carte, etc.). Le mécanisme de transaction permet de s'assurer que toutes les opérations ou aucune opération dans la transaction est terminée. De cette façon il est possible de maintenir la cohérence interne des données connexes.

Cependant, ce mécanisme introduit une vulnérabilité à l'environnement d'exécution Java Card (présentée dans la section 3.1.3) qui permet de réaliser des attaques d'accès (en lecture et écriture) en mémoire EEPROM complète d'une Java Card.

Les travaux de [6] ont montré que malgré la présence des mécanismes de protection utilisés par Java Card, cette attaque a pu être réalisée sur certains types de cartes (des cartes ouvertes possédant certains mécanismes de protection).

4.2. Le problème avec les mécanismes de protection

Nous avons déjà présenté les mécanismes de protection, contre les attaques de confusion de type, utilisés par Java Card (dans le Chapitre 4). La carte dispose de deux importants mécanismes de protection contre ce type d'attaques : le vérifieur de bytecode (sur carte ou bien hors-carte utilisant une signature numérique) et le pare-feu.

Nous avons déjà discuté le but de la vérification de bytecode dans le Chapitre 4. Le vérifieur de bytecode a été prouvé insuffisant pour le type d'attaques que nous avons présenté, ici. Ceci est parce que le code produit par le compilateur, transformé, et exécuté sur la carte n'a pas été modifié et il est parfaitement légal de point de vue du vérifieur.

Le pare-feu (présenté dans le chapitre 4) permet de mettre en œuvre une politique d'isolation des applets et de contrôle d'interactions entre ces applets. Il opère lors de chaque accès mémoire à l'exécution et impose une stricte séparation entre les espaces mémoire des différentes applets présentes sur la carte. Cela devrait fournir un mécanisme de protection de la carte et des applets installées sur cette carte. Cependant, le problème générique avec le pare-feu, c'est qu'il protège les données par la référence de Java (ou bien du bytecode) et non par le pointeur de mémoire réel. C.à.d. que c'est l'accès à la référence qui est limitée pour une applet donnée et non pas l'accès au bloc de mémoire que cette référence pointe. Lorsque la référence est légale (appartenant à l'applet qui veut y accéder), mais le pointeur ne l'est pas, le pare-feu ne détecte pas l'accès à l'emplacement mémoire non autorisé.

En raison de la conception insuffisante et de la présence de bugs (vulnérabilités) dans l'implémentation de l'environnement d'exécution de la carte, les deux mécanismes mentionnés ne peuvent pas protéger la carte contre l'attaque que nous avons présenté, ici.

D'après [6], ils existent des Java Cards ouvertes qui ont pu résister à ce type d'attaques. La principale raison de résistance de ces cartes à l'attaque c'est qu'elles possèdent des mécanismes de protection supplémentaires (autres que le vérifieur et le pare-feu). En particulier, ils effectuent des vérifications lors de l'exécution pour éviter dynamiquement toute sorte de confusion de type. Le fonctionnement exact de ces vérifications dynamiques est très difficile à examiner sur les Java Card. Ces vérifications peuvent aller de simples contrôles d'intégrité sur les références à une vérification complète dynamique de typage.

Nous avons discuté, dans le Chapitre 4, la différence entre la vérification de code statique et dynamique, et nous avons jugé la vérification statique comme étant la meilleure rapproche de vérification de point de vue de la sécurité ou bien du coût de la vérification. [6]

4.3. Solution proposée

Au sein de ce mémoire, nous avons essayé de réaliser une contre mesure pour faire face aux attaques de confusion de type qui sont produites à travers l'utilisation de la vulnérabilité présente dans le mécanisme de transaction offert par l'environnement d'exécution Java Card.

L'exploitation de cette vulnérabilité permet de réaliser des attaques qui utilisent des références de données légales au niveau du code Java (et du bytecode) mais pas au niveau des pointeurs de mémoire réels.

Notre solution consiste à vérifier, au niveau du bytecode (contenu dans le fichier CAP), la présence de la séquence d'instructions qui permet de produire une telle confusion de type. Comme cette vérification va s'effectuer sur le fichier CAP, nous avons pensé qu'il serait préférable d'introduire un module, qui effectue cette vérification, au niveau du vérifieur statique de bytecode Java Card, qui lui aussi effectue la vérification sur le fichier CAP.

4.4. Réalisation du nouveau vérifieur

Nous allons utiliser le kit de développement pour la plate-forme Java Card 2.2.2 (Javacard Development Kit 2.2.2) pour appliquer notre solution. Rappelons, ici, que cet environnement simule une Java Card qui possède la plate-forme Java Card d'exécution d'applets, version 2.2.2, et offre les différents outils nécessaires pour les transformations d'applets et leur chargement et exécution sur la carte. Cet environnement offre une plate-forme Java Card qui repose sur la vérification du fichier CAP hors-carte, et offre l'outil de la vérification du fichier CAP hors-carte `verifycap`.

Nous avons décrit le rôle et le fonctionnement de l’outil `verifycap` dans la section 2.2.5. Il permet de vérifier les contraintes sur le bytecode contenu dans le fichier CAP afin de détecter les erreurs de confusion de type. Nous avons prouvé, dans la section 3.2.2, l’inefficacité de ce vérifieur pour détecter la confusion de type réalisée au niveau des pointeurs de mémoire réels (mais qui utilise des références de données légalées au niveau du bytecode) et produite à travers l’utilisation de la vulnérabilité du mécanisme de transaction.

Notre travail consiste à introduire le nouveau module de vérification que nous avons décrit précédemment à l’outil `verifycap`, et prouver la réussite, du nouveau vérifieur produit, à détecter la confusion de type présentée, ici.

Pour réaliser ce travail, nous avons suivi les étapes décrites dans les sections suivantes.

4.4.1. Accéder au code source du vérifieur

Le kit de développement Javacard Development Kit 2.2.2 offre un ensemble d’outils qui sont écrits à la base en langage Java et fournis en code intermédiaire contenu dans des fichiers JAR. Ces outils utilisent, pour leur fonctionnement en ligne de commande, des commandes qui sont écrites en langage de script (shell). Après l’installation de cet environnement, des dossiers contenant tous les outils et les bibliothèques nécessaires à son fonctionnement, sont créés dans notre système. Dans ces dossiers, nous trouverons les fichiers de script (.bat) et les fichiers JAR nécessaires au fonctionnement de chaque outil offert par cet environnement.

Pour introduire notre nouveau module de vérification, nous allons modifier dans le contenu du fichier de script et de fichier JAR correspondants au vérifieur `verifycap`, à savoir, `verifycap.bat` et `offcardverifier.jar`.

La modification dans le contenu du fichier de script `verifycap.bat` est facile à effectuer, il suffit d’ouvrir le fichier dans un éditeur de texte et le modifier. La complication se situe dans la modification dans le fichier JAR.

En premier lieu, il faut réussir à ouvrir le fichier JAR et à lire son contenu. Le fichier JAR est un container, il regroupe un ensemble de fichiers class. On peut ouvrir le fichier JAR on le traitant comme une archive compressée, il suffit de le dézipper pour pouvoir accéder aux fichiers class qu’il contient.

Le format des fichiers class est un format illisible pour l’utilisateur. Pour pouvoir lire le contenu de ces fichiers nous avons utilisé le décompilateur Java **Cavaj**.

Cavaj est un logiciel qui permet la décompilation d’un fichier class. La compilation d’un fichier Java permet de transformer ce fichier en fichier class. La décompilation est l’opération inverse, c.à.d. elle permet de transformer un fichier class en un fichier Java (obtenir le code source depuis lequel le fichier class à été généré).

L'utilisation de ce logiciel nous a permis d'accéder au code source de l'outil `verifcap` écrit en Java.

4.4.2. Ajouter le nouveau module de vérification

Maintenant, que nous avons le code source du vérifieur, nous pouvons modifier dans ce code, et donc nous pouvons ajouter le nouveau module de vérification à ce code. Comme le code du vérifieur est écrit en langage Java, nous allons, bien évidemment, écrire les fonctions que nous allons développer dans ce nouveau module de vérification en langage Java, et les introduire dans le code du vérifieur.

L'analyse de ce code nous a permis de mieux comprendre le fonctionnement interne du vérifieur. Nous avons remarqué qu'il opère sur les instructions du fichier CAP directement, instruction par instruction. Nous voulions suivre la même politique de vérification effectuée par le vérifieur pour développer nos fonctions de vérifications, c.à.d. opérer directement sur les instructions du fichier CAP. Cependant, nous avons rencontré un problème pour faire cela. Le problème est que le format du fichier CAP est illisible par l'utilisateur, et donc nous ne pourrions pas analyser des instructions que nous ignorons leur structure.

La solution à ce problème était d'effectuer la vérification sur le fichier au format JCA et non pas au format CAP. Nous rappelant, ici, la définition du format JCA, qui est une représentation textuelle lisible d'un fichier CAP qu'on peut utiliser pour faciliter les tests et le débogage.

Le fichier JCA contient les mêmes informations que contient le fichier CAP. Les instructions contenues dans ce fichier sont lisibles, et donc, nous pouvons opérer sur ces instructions et les analyser sans aucun problème. La figure 6.19 montre un échantillon de code contenu dans le fichier JCA correspondant à l'applet `MonAttack` (les instructions sélectionnées représentent une partie des instructions malveillantes correspondantes au code présenté dans la section 3.1.3).

L'utilisation du fichier JCA au lieu du fichier CAP a causé une légère modification dans la conception du vérifieur. Au lieu de prendre le fichier CAP comme paramètre d'entrée, on doit prendre le fichier JCA.

```

invokevirtual 10; // setOutgoingAndSend(SS)V
aload 0;
sconst_2;
newarray 11;
putfield_a 3; // reference packageAttack/MonAttack.arrayMutable
aconst_null;
checkcast 12 0; // T_SHORT
astore 3;
invokestatic 12; // javacard/framework/JCSystem.beginTransaction()V
aload 0;
sconst_1;
newarray 12;
putfield_a 4; // reference packageAttack/MonAttack.arrayS
getfield_a_this 4; // reference packageAttack/MonAttack.arrayS
astore_3;
invokestatic 14; // javacard/framework/JCSystem.abortTransaction()V
aload 0;
sspush 128;
newarray 11;
putfield_a 5; // reference packageAttack/MonAttack.arrayB

```

Figure 6.19. Echantillon du code du fichier JCA.

- **Description du nouveau vérifieur**
 - Le nouveau vérifieur prend un fichier JCA comme paramètre d'entrée.
 - La première fonction appelée par le vérifieur est la fonction principale du module de vérification que nous avons ajoutée.
 - Cette fonction permet de lire le fichier JCA et de rechercher les blocs d'instructions qui commencent avec l'instruction `JCSystem.beginTransaction()` et se terminent avec l'instruction `JCSystem.abortTransaction()`. Pour chaque bloc trouvé, on analyse la séquence d'instructions contenue dans ce bloc, et on vérifie si après l'exécution de ces instructions, un pointeur à la mémoire EEPROM va être sauvegardé dans une variable locale. Si c'est le cas, ça veut dire qu'on a trouvé une confusion de type qui peut être exploitée pour construire une attaque, alors on relève une exception, on affiche l'erreur correspondante et on arrête la vérification.
 - Sinon, on continue la vérification. On appelle l'outil capgen pour générer un fichier CAP à partir du fichier JCA entré en paramètre. Et on appelle la fonction principale de l'ancien vérifieur en lui donnant comme paramètre d'entrée, le fichier CAP généré.
 - Le vérifieur effectue la vérification habituelle du fichier CAP, et affiche s'il contient des erreurs ou non.

4.4.3. Reproduire le vérifieur

Maintenant, que nous avons la structure générale du nouveau vérifieur, nous allons rédiger son code en langage Java. Pour cela, nous utilisons l'environnement de développement Java Eclipse SDK, version 3.2.2.

Cet environnement permet de créer un nouveau projet Java à partir des fichiers class existants et d'ajouter d'autres fichiers à ce projet.

Nous avons utilisé cet environnement pour créer un nouveau projet dans lequel nous avons construit notre nouveau vérifieur. Nous avons ajouté tous les fichiers class et de descriptions contenues dans le fichier JAR de l'ancien vérifieur à ce projet. Après, nous avons créé une nouvelle classe dans ce projet dans laquelle nous avons écrit le code du nouveau module de vérification, ainsi que les fonctions qui permettent d'établir les liens avec les autres classes.

Ensuite, nous avons changé la fonction principale du projet par la fonction principale de cette nouvelle classe ajoutée. La figure 6.20 montre la structure de notre projet (appelé *offcardverifier*) dans Eclipse et présente un échantillon du code de notre nouvelle classe (appelée *NouvVerification*).

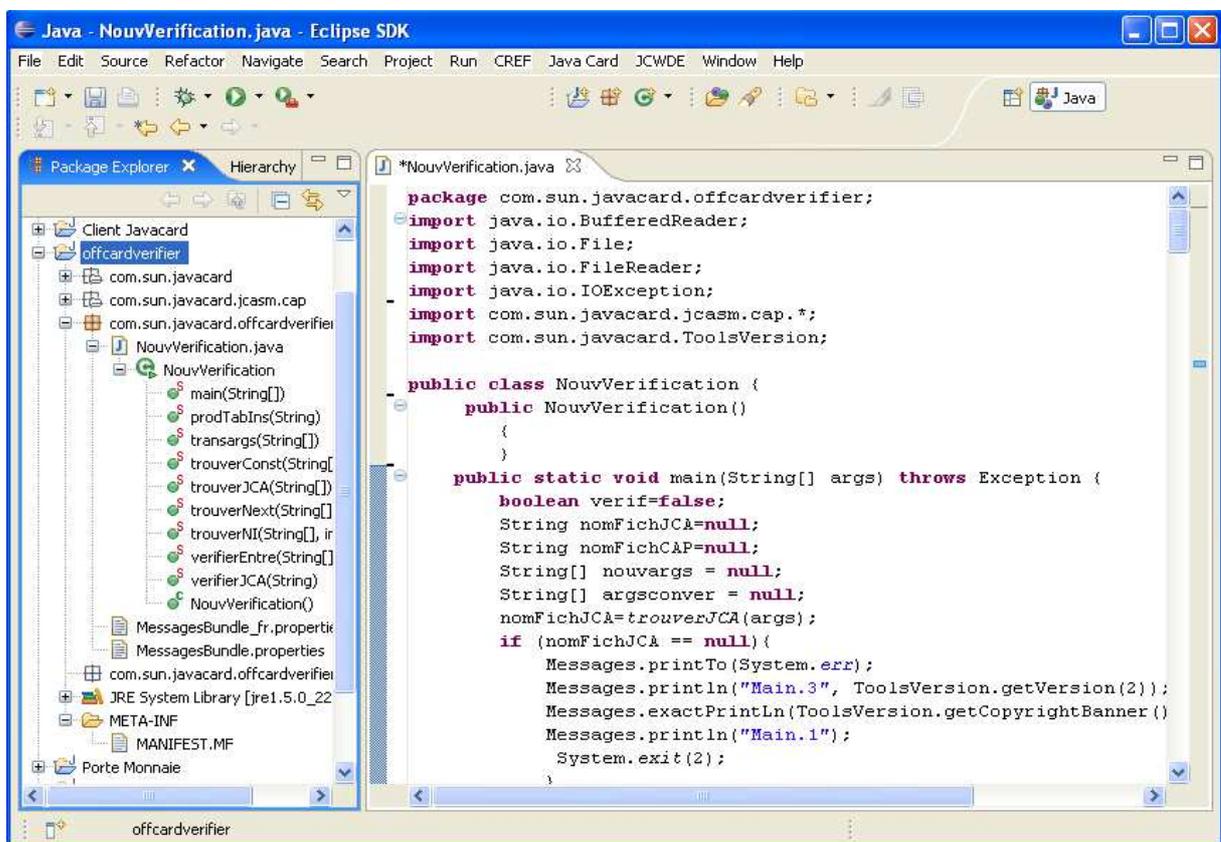


Figure 6.20. Projet *offcardverifier* sous Eclipse.

Maintenant, que le projet est construit, il suffit d'exporter ce projet en un fichier JAR, pour qu'on puisse l'exécuter, en lui donnant le même nom et la même organisation interne que l'ancien vérifieur. Et ensuite, remplacer le fichier JAR *offcardverifier.jar* qui est situé dans la bibliothèque du kit de développement (Javacard Development Kit 2.2.2) par celui que nous avons produit.

Notons, ici, que nous devons modifier aussi dans le fichier de script *verifycap.bat*, qui est utilisé pour lancer le vérifieur *verifycap* en lignes de commandes, afin de l'adapter au fonctionnement du nouveau vérifieur.

4.4.4. Résultats de l'utilisation du nouveau vérifieur

L'utilisation du nouveau vérifieur ressemble à celle de l'ancien vérifieur (décrite dans la section 2.4.3). Pour vérifier un fichier CAP avec notre nouveau vérifieur, on lance l'outil *verifycap* en ligne de commandes et on entre en paramètre le nom du fichier JCA correspondant à notre fichier CAP (à la différence avec l'ancien vérifieur, qui prend en paramètre d'entrée le nom du fichier CAP). Le vérifieur effectue la fonction de vérification habituelle, en plus de la nouvelle fonction de vérification qu'on a ajoutée, et affiche le résultat de la vérification.

Nous allons, maintenant, tester l'efficacité de ce nouveau vérifieur à détecter les attaques de confusion de type habituelles et celles qu'on a décrit dans ce chapitre.

Commençons par les attaques de confusion de type, produites à travers l'utilisation de la vulnérabilité du mécanisme de transaction, que nous avons décrit, ici, (où la confusion de type est réalisée au niveau des pointeurs de mémoire réels et non pas au niveau des références de données dans le bytecode). Nous allons vérifier le fichier CAP *packageAttaque.cap*, qui contient une attaque de ce type (décrit dans la section 3.2.2), par notre nouveau vérifieur et voir le résultat de cette vérification.

Pour effectuer cette vérification, nous lançons l'outil *verifycap* en ligne de commandes et nous lui donnons comme paramètre d'entrée le fichier JCA correspondant au fichier CAP que nous voulons vérifier *packageAttaque.jca*. Le résultat de la vérification est montré dans la figure 6.21.

```
G:\Documents and Settings\yousra>
G:\Documents and Settings\yousra>verifycap "G:\JCDK\java_card_kit-2_2_2\api_export_files\java\lang\javacard\lang.exp" "G:\JCDK\java_card_kit-2_2_2\api_export_files\javacard\framework\javacard\framework.exp" "G:\JCDK\java_card_kit-2_2_2\api_export_files\javacard\crypto\javacard\crypto.exp" "G:\JCDK\java_card_kit-2_2_2\api_export_files\javacard\security\javacard\security.exp" "workspace\Attack\bin\packageAttack\javacard\packageAttack.jca"
1.3 Off-Card Verifier, Version <1>.
Copyright 2005 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms.
Vérification du fichier JCA workspace\Attack\bin\packageAttack\javacard\packageAttack.jca
Erreur de type: potentielle confusion de type au niveau des pointeurs de la mémoire EEPROM, causée par le mécanisme de transaction
La vérification est terminée, 1 erreur.
G:\Documents and Settings\yousra>
```

Figure 6.21. Résultat de la vérification avec le nouveau vérifieur.

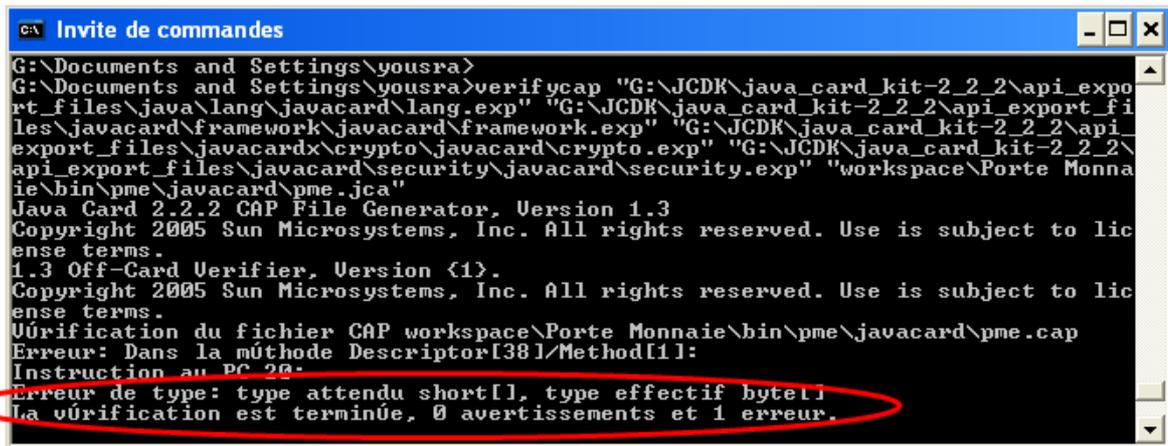
Nous remarquons, ici, que le vérifieur affiche une erreur d'une potentielle confusion de type. Sa veut dire qu'il a détecté le pointeur de mémoire sauvegardé dans la variable locale, qui peut être utilisé pour créer une confusion de type.

Nous allons prouver que ce vérifieur ne bloque pas les autres instructions du mécanisme de transaction. Si on enlève du code de l'applet MonAttack, l'instruction qui permet de sauvegarder le pointeur dans la variable locale (sans cette instruction, on ne peut pas effectuer l'attaque), le vérifieur ne signale aucune erreur, et donc on peut utiliser le mécanisme de transaction normalement. Le résultat de la vérification est montré dans la figure 6.22.

```
G:\Documents and Settings\yousra>
G:\Documents and Settings\yousra>verifycap "G:\JCDK\java_card_kit-2_2_2\api_export_files\java\lang\javacard\lang.exp" "G:\JCDK\java_card_kit-2_2_2\api_export_files\javacard\framework\javacard\framework.exp" "G:\JCDK\java_card_kit-2_2_2\api_export_files\javacard\crypto\javacard\crypto.exp" "G:\JCDK\java_card_kit-2_2_2\api_export_files\javacard\security\javacard\security.exp" "workspace\Attack\bin\packageAttack\javacard\packageAttack.jca"
Java Card 2.2.2 CAP File Generator, Version 1.3
Copyright 2005 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms.
1.3 Off-Card Verifier, Version <1>.
Copyright 2005 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms.
Vérification du fichier CAP workspace\Attack\bin\packageAttack\javacard\packageAttack.cap
La vérification est terminée, 0 avertissements et 0 erreurs.
G:\Documents and Settings\yousra>
```

Figure 6.22. Résultat de la vérification après modification.

Nous allons, maintenant tester l'efficacité du vérifieur à trouver les erreurs de confusion de type habituelles. Nous vérifions le fichier **pme.cap** après l'ajout de l'attaque de confusion de type (présenté dans la section 2.4.3). Le résultat de la vérification est la détection de la confusion de type. La figure 6.23 montre cela.



```

C:\> Invite de commandes
G:\Documents and Settings\yousra>
G:\Documents and Settings\yousra>verifycap "G:\JCDK\java_card_kit-2_2_2\api_export_files\java\lang\javacard\lang.exp" "G:\JCDK\java_card_kit-2_2_2\api_export_files\javacard\framework\javacard\framework.exp" "G:\JCDK\java_card_kit-2_2_2\api_export_files\javacard\crypto\javacard\crypto.exp" "G:\JCDK\java_card_kit-2_2_2\api_export_files\javacard\security\javacard\security.exp" "workspace\Porte Monnaie\bin\pme\javacard\pme.jca"
Java Card 2.2.2 CAP File Generator, Version 1.3
Copyright 2005 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms.
1.3 Off-Card Verifier, Version {1}.
Copyright 2005 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms.
Vérification du fichier CAP workspace\Porte Monnaie\bin\pme\javacard\pme.cap
Erreur: Dans la méthode Descriptor[38]/Method[1]:
Instruction au PC 20:
Erreur de type: type attendu short[], type effectif byte[]
La vérification est terminée, 0 avertissements et 1 erreur.

```

Figure 6.23. Résultat de la vérification du fichier pme.cap contenant l'attaque.

5. DISCUSSION

Nous avons réalisé, au sein de ce mémoire, un vérifieur statique hors-carte du fichier CAP à partir de celui fourni par le kit de développement (Javacard Development Kit 2.2.2).

Le problème avec l'ancien vérifieur c'est qu'il ne détecte pas les erreurs de confusion de type qui sont réalisées au niveau des pointeurs de mémoire réels et non pas au niveau des références de données dans le bytecode. Ce genre de confusion de type peut être réalisé à travers l'utilisation de la vulnérabilité présente dans le mécanisme de transaction offert par la plate-forme Java Card. D'après les travaux de [6] on peut créer une telle confusion de type et réaliser des attaques d'accès à la mémoire EEPROM complète d'une Java Card, pour certains types de cartes.

Nous avons utilisé le kit de développement (Javacard Development Kit 2.2.2) pour simuler une Java Card qui a les mêmes caractéristiques des cartes qu'on peut attaquer (cartes ouvertes sans mécanismes de protection additionnelles). Nous avons, ainsi, utilisé les différents outils offerts par cet environnement, en particulier le vérifieur hors-carte, pour démontrer les résultats de vérification de fichiers CAP.

Nous avons développé un module de vérification qui permet de détecter ce genre de confusion de type au niveau des instructions du fichier JCA (la version lisible du fichier CAP). Et nous l'avons introduit au vérifieur offert par le kit.

Nous avons prouvé l'efficacité de ce vérifieur à travers les résultats présentés dans la section précédente. Rappelons, ici, que le vérifieur utilisé par le simulateur n'affecte pas une signature numérique au fichier vérifié, il affiche juste le résultat de la vérification. C'est au développeur de décider de charger le fichier sur la carte ou pas. Ce n'est pas le cas, dans une carte réelle ou on vérifie la signature du fichier au moment du chargement. Et donc si le fichier contient des erreurs on ne peut pas le charger sur la carte.

En conclusion notre nouveau vérifieur permet de faire face à certain type d'attaques qu'on peut réaliser sur un certain type de cartes à puce. Par exemple l'attaque que nous avons présentée dans la section 3.2 et qui permet de réaliser un gain financier sur une carte de crédit.

La réalisation de la vérification au niveau du fichier CAP et son introduction dans l'ancien vérifieur offre les avantages suivants :

- Effectuer la vérification du code au niveau du bytecode contenu dans le fichier CAP est beaucoup plus facile que l'effectuer dynamiquement au moment de l'exécution et au niveau des pointeurs mémoire réels. En même temps cette vérification peut détecter la confusion de type au niveau des pointeurs mémoire réels (c.à.d. l'accès aux emplacements mémoires non autorisées) même si la référence au niveau du bytecode est légale.
- La réalisation de la vérification du bytecode statiquement, et donc ne pas avoir besoin de la vérification dynamique, qui est chère et qui ralentit l'exécution de manière significative.
- L'introduction du nouveau module de vérification dans le système Java Card ne modifie que dans le vérifieur du bytecode. La conception du système d'exploitation (environnement d'exécution) Java Card et des autres mécanismes de protection reste elle-même. Et donc on n'affecte pas les fonctionnalités offertes par la carte et le fonctionnement des mécanismes de protection contre les différents types d'attaques (par exemple, le pare-feu).
- Dans le cas du vérifieur hors-carte, notre modification dans le vérifieur est séparée complètement de la carte. La vérification peut fonctionner de la même manière qu'avant, c.à.d. attribuer une signature au fichier CAP qui sera vérifiée par la carte lors du chargement du fichier sur cette carte. Par conséquent, l'introduction de ce nouveau module de vérification peut affecter mêmes les cartes qui sont déjà délivrées ou bien qui sont dans le marché (on n'a pas besoin de produire de nouvelles cartes pour leurs introduire ce nouveau module de vérification).

Cependant, ils existent des inconvénients qui sont dus à la vérification au niveau du fichier CAP, l'utilisation du fichier JCA, et l'utilisation du simulateur :

- La vérification au niveau du fichier CAP (ou JCA) ne permet pas de vérifier la présence de la confusion de type dans les pointeurs de mémoire réels, mais de vérifier la présence de la séquence d'instructions qui produit cette confusion de type. Donc, si on arrive à causer cette confusion de type à travers un autre moyen (autre que le mécanisme de transaction), notre vérifieur ne pourra pas détecter l'erreur.
- Effectuer la vérification sur le fichier JCA et non pas sur le fichier CAP exige une tâche supplémentaire à l'utilisateur. A savoir trouver le moyen d'obtenir le fichier JCA (utiliser des différents logiciels et outils de transformation), ce qui est une tâche un peu lourde pour l'utilisateur.
- L'utilisation du simulateur au lieu d'une carte réelle présente des inconvénients. C'est toujours mieux de tester sur une carte réelle que sur un simulateur, des problèmes qui ne sont pas rencontrés avec le simulateur peuvent surgir dans la carte réelle. Nous avons vu que le simulateur ne supporte pas la fonctionnalité de la signature électronique du fichier vérifié et que le fonctionnement interne de la mémoire réelle est différent de celui du simulateur. Ceci peut engendrer des problèmes dans l'implémentation de notre solution dans une carte réelle qu'on ne peut pas les découvrir par l'utilisation du simulateur.

6. CONCLUSION

Nous avons présenté, dans ce chapitre, une attaque de confusion de type qui n'est pas détectée par le vérifieur de fichier CAP hors-carte fourni par SUN. Nous avons démontré cela sur le vérifieur offert par l'environnement de développement pour la plate-forme Java Card 2.2.2 (Javacard Development Kit 2.2.2).

Nous avons offert une solution à ce problème qui consiste au développement d'un module de vérification qui permet l'analyse statique du code contenu dans le fichier CAP (au format JCA). Ce module de vérification permet de détecter les attaques de confusion de type présentées, ici, et qui ne sont pas détectées par l'ancien vérifieur. Ensuite, nous avons introduit ce nouveau module de vérification dans le vérifieur offert par l'environnement de développement que nous avons utilisé et reproduit un nouveau vérifieur de fichier CAP. Nous avons démontré à travers des exemples et des résultats de vérification l'efficacité de ce nouveau vérifieur et son capacité à détecter les attaques de confusion de type qui ne sont pas détectés par le vérifieur de fichier CAP hors-carte de SUN.

Nous avons vu que cette solution de sécurité apporte beaucoup d'avantages aux cartes à puces. Par exemple, la réalisation de la vérification du bytocode statiquement, et donc ne pas avoir besoin de la vérification dynamique, qui est chère et qui ralentit l'exécution de manière significative. En plus, elle permet de garder les fonctionnalités de la carte et des autres mécanismes de protection, et dans le cas du vérifieur hors-carte, nous n'aurons pas besoin à reproduire les cartes pour qu'elles utilisent le nouveau vérifieur, nous pourrions utiliser ce nouveau vérifieur avec les anciennes cartes.

Nous avons vu, aussi, que cette solution présente un inconvénient au niveau de la sécurité. La vérification au niveau du fichier CAP (ou JCA) ne permet pas de vérifier la présence de la confusion de type dans les pointeurs de mémoire réels, mais de vérifier la présence de la séquence d'instructions qui produit cette confusion de type. Donc, si on arrive à causer cette confusion de type à travers un autre moyen (autre que le mécanisme de transaction), notre vérifieur ne pourra pas détecter l'erreur.

Nous concluons, ici, que malgré les différents mécanismes de sécurité qu'on peut développer, un attaquant peut toujours trouver un moyen pour passer ces mécanismes et réaliser des attaques sur le système. La mise en œuvre des mécanismes de protection ne permet pas d'assurer la sécurité du système totalement, mais elle permet de diminuer le risque d'être attaqué en faisant face à autant de types d'attaques que possible, et nous rendrons, donc, la tâche d'un attaquant difficile à réaliser.

Conclusion et perspectives

Conclusion et perspectives

Assurer la sécurité dans les cartes à puce, équipées avec une plate-forme Java Card, est une tâche très importante. Ceci est dû à la sensibilité des données qui peuvent être contenues dans ces cartes (codes d'accès aux comptes, informations personnelles, données médicales, etc.) et aux contraintes liées aux dispositifs matériels (les ressources limitées et les contraintes de puissance) qui peuvent être un obstacle à la mise en œuvre des mécanismes de protections.

Nous avons traité, dans ce mémoire les problèmes de sécurité qui parviennent du chargement d'applications externes sur les Java Cards après leur délivrance. Plus précisément, l'introduction d'un code malicieux contenant une confusion de type dans le bytecode contenu dans un fichier CAP. Le chargement de ce fichier dans la carte permet de nuire à son fonctionnement et peut attaquer les autres applications présentes sur la carte.

Nous avons présenté un état de l'art des mécanismes de protections mises en œuvre par les Java Cards à fin de faire face à ce type d'attaques. Particulièrement, le vérifieur statique de fichier CAP fournis par SUN Microsystems qui représente le point clé de ces mécanismes. Ce vérifieur permet de vérifier si un code donné respecte les contraintes imposées sur le bytecode Java Card ou non. Il est utilisé juste avant le chargement d'une application sur la carte, et il permet de détecter les erreurs de confusion de type présentes dans ce code et interdire le chargement de ce code dans la carte à puce dans le cas où il en trouve des erreurs.

Malgré la puissance et l'efficacité de ce vérifieur à détecter les différentes erreurs de confusion de type, il existe une vulnérabilité dans le système Java Card qui permet à une application contenant un code malicieux (une confusion de type) de passer ce vérifieur et se charger sur la carte. Cette vulnérabilité est présente au niveau du mécanisme de transaction offert par la plate-forme Java Card. Ce mécanisme est utilisé pour protéger la cohérence des données persistantes stockées sur la carte dans la mémoire EEPROM. Cependant, il introduit une vulnérabilité à l'environnement d'exécution Java Card qui permet de réaliser des attaques d'accès (en lecture et écriture) en mémoire EEPROM complète de la Java Card.

L'exploitation de cette vulnérabilité permet de réaliser des attaques qui utilisent des références de données légales au niveau du code Java (et du bytecode) mais pas au niveau des pointeurs de mémoire réels. Le vérifieur de SUN a été prouvé insuffisant pour ce type d'attaques. Ceci est parce que le code produit par le compilateur, transformé, et exécuté sur la carte n'a pas été modifié et il est parfaitement légal de point de vue du vérifieur.

Nous avons proposé, au sein de ce mémoire, une solution qui permet de faire face à ce type d'attaques. Notre solution consiste à vérifier, au niveau du bytecode contenu dans le fichier CAP, la présence de la séquence d'instructions qui permet de produire la confusion de type précédente. Cette solution a été proposée du point de vue que la réalisation de la vérification du code au niveau du fichier CAP est beaucoup plus facile et moins chère que la vérification du code dynamique au moment de l'exécution et au niveau des pointeurs mémoire réels. En même temps cette vérification peut détecter la confusion de type au niveau des pointeurs mémoire réels (c.à.d. l'accès aux emplacements mémoires non autorisées) même si la référence au niveau du bytecode est légale.

Nous avons réalisé une contre mesure sécuritaire qui consiste au développement d'un nouveau module de vérification de fichier CAP et son introduction dans le vérifieur statique de fichier CAP de SUN.

Nous avons utilisé l'environnement de développement pour la plate-forme Java Card 2.2.2 (Javacard Development Kit 2.2.2) et l'environnement de développement Eclipse, pour simuler le fonctionnement d'une Java Card (qui a certaines caractéristiques) et pour le développement et l'exécution des différentes applications utilisées au sein de ce travail. Nous avons utilisé l'environnement Eclipse pour développer notre nouveau module de vérification (écrit en Java) et nous l'avons introduit dans l'outil de vérification offert par l'environnement de développement pour la plate-forme Java Card 2.2.2 (Javacard Development Kit 2.2.2). Cet environnement nous a permis d'obtenir les résultats de la vérification avec notre nouveau vérifieur et de les comparer avec celles de l'utilisation de l'ancien vérifieur.

D'après les résultats que nous avons obtenu, notre nouveau vérifieur est prouvé efficace à détecter les erreurs de confusion de type présentées ici, et qui ne sont pas détectées par l'ancien vérifieur. Il permet de vérifier les instructions du mécanisme de transaction qui peuvent causer une attaque mais ne bloque pas ce mécanisme, c.à.d. que si la séquence d'instructions vérifiée n'est pas suffisante pour réaliser une attaque, le vérifieur ne signale pas une erreur. Ce nouveau vérifieur permet, aussi, de détecter les erreurs de confusion de type habituelles qui peuvent être détectées par l'ancien vérifieur.

La solution de sécurité que nous avons proposée et réalisée au sein de ce mémoire, offre beaucoup d'avantages aux cartes à puces par rapport aux solutions existantes. Parmi ses avantages : elle permet de garder les fonctionnalités de la carte et des autres mécanismes de protection utilisés par la carte, et dans le cas du vérifieur hors-carte, nous n'aurons pas besoin à reproduire les cartes pour qu'elles utilisent le nouveau vérifieur, nous pourrons utiliser ce nouveau vérifieur avec les anciennes cartes.

Cependant, cette solution présente un inconvénient au niveau de la sécurité. La vérification au niveau du fichier CAP ne permet pas de vérifier la présence de la confusion de type dans les pointeurs de mémoire réels, mais de vérifier la présence de la séquence d'instructions qui produit cette confusion de type. Donc, si on arrive à causer cette confusion de type à travers un autre moyen (autre que le mécanisme de transaction), notre vérifieur ne pourra pas détecter l'erreur.

Nous avons rencontré, aussi, quelques problèmes. Le premier est du au format illisible du fichier CAP, donc, nous avons réalisé la vérification sur le fichier JCA (version lisible du fichier CAP), ceci exige une tâche supplémentaire à l'utilisateur. A savoir trouver le moyen d'obtenir le fichier JCA (utiliser des différents logiciels et outils de transformation), ce qui est une tâche un peu lourde pour l'utilisateur. Le deuxième problème est du à l'utilisation d'un simulateur de carte au lieu d'une carte réelle. Le simulateur n'offre pas certaines fonctionnalités de la carte réelle, par exemple, la signature numérique. Ceci peut engendrer des problèmes dans l'implémentation de cette solution dans une carte réelle qu'on ne peut pas les découvrir par l'utilisation du simulateur.

À l'issue de ce travail, ces problématiques nous semblent mériter d'être approfondies. Nos perspectives pour des travaux futurs sont de traiter ces problèmes. En premier lieu, nous souhaitons effectuer une étude approfondie sur le format du fichier CAP et trouver un moyen de réaliser la vérification sur les instructions de ce fichier. Un autre point intéressant est d'attribuer une signature numérique, séparée de celle du vérifieur, à notre nouveau module de vérification et donc, l'utilisation d'une carte réelle à fin de pouvoir tester les résultats.

Bibliographie

Bibliographie

- [1] Marc Eluard ; *Analyse de sécurité pour la certification d'applications Java Card* ; thèse de doctorat ; université de Rennes 1, IRISA, Ecole doctorale MATISSE; 10 décembre 2001.
- [2] Yves Deswarte ; *Comment mesurer la sécurité informatique ?* ; rapport technique ; LAAS, Laboratoire d'Analyse et d'Architecture des Systèmes, CNRS ; 2000.
- [3] Laurent Bloch, Christophe Wolfhugel ; *Sécurité Informatique : Principes et méthode* ; livre ; ÉDITIONS EYROLLES ; 7 Mai 2009 ; p. 7-86.
- [4] Guillaume Dufay ; *Vérification formelle de la plate-forme Java Card* ; thèse de doctorat ; Université de Nice - SOPHIA ANTIPOLIS UFR Sciences, Ecole doctorale STIC ; 5 décembre 2003.
- [5] Lilian Burdy, Ludovic Casset, et Antoine Requet ; *Développement formel d'un vérifieur embarqué de byte-code Java* ; article ; Publié dans Technique et Science Informatiques (TSI) 22 ; 2003.
- [6] Jip Hogenboom, Wojciech Mostowski ; *Full Memory Read Attack on a Java Card* ; article ; Proceedings of 4th Benelux Workshop on Information and System Security, Louvain-la-Neuve, Belgium ; Novembre 2009.
- [7] *Introduction à la sécurité informatique.*
<http://www.commentcamarche.net/contents/secu/secuintro.php3> ; 2011.
- [8] Richard A. Kemmerer ; *An introduction to computer security* ; rapport technique ; Computer Science Department, University of California, Santa Barbara, California, U.S.A. ; 2010.
- [9] Wikipédia ; *Sécurité du système d'information.*
http://fr.wikipedia.org/wiki/Sécurité_du_système_d'information ; 2011.
- [10] Gérard - Michel Cochard ; *Virus, vers et cie* ; Chapitre 6 du cours du Module D312 (Sécurité des systèmes d'information), MASTER 2 E-MIAGE (UPJV) ; 2011.
- [11] Therrezinha Fernandes ; *Les politiques de sécurité* ; rapport technique ; CRSNG (Conseil de recherches en sciences naturelles et en génie du Canada) ; 25 mai 2001.

- [12] Dominique Mouchéné ; *Sécurité informatique* ; rapport technique ; IAI Savoie ; juin 2005.
- [13] Raphael Khoury ; *Détection du code malicieux : système de type à effets et instrumentation du code* ; mémoire de Maître ès sciences ; Faculté des sciences et de génie, université Laval, Québec ; 2005.
- [14] Pierre-Alain Fouque ; *Cryptographie appliquée* ; rapport technique ; Laboratoire de cryptographie de la direction centrale de la Sécurité des systèmes d'information (DCSSI) ; 2004.
- [15] Wikipédia ; *Cryptographie*.
[http:// fr.wikipedia.org/wiki/ Cryptographie](http://fr.wikipedia.org/wiki/Cryptographie) ; 2011.
- [16] Chikouche Nouredine ; *Problèmes de sécurité dans les systèmes embarqués* ; mémoire de magistère ; Université de M'sila, Faculté de Mathématiques et Informatique, Département d'Informatique ; 23 septembre 2010.
- [17] Djellab Rima ; *Les systèmes embarqués et le temps réel* ; rapport d'études ; Département informatique, Faculté des sciences, Université El Hadj Lakhder, Batna ; 2008.
- [18] Lyes Khelladi, Yacine Challal, Abdelmadjid Bouabdallah, et Nadjib Badache ; *On Security Issues in Embedded Systems : Challenges and Solutions* ; article ; International Journal of Information and Computer Security ; 2008 ; p. 140-174.
- [19] Wikipédia ; *Système embarqué*.
http://fr.wikipedia.org/wiki/Système_embarqué ; 2011.
- [20] Patrice Kadionik ; *Les systèmes embarqués : une introduction* ; rapport technique ; Ecole Nationale Supérieure d'Electronique, Informatique, Télécommunications, Mathématique et Mécanique de Bordeaux (ENSEIRB) ; Décembre 2005.
- [21] Alexandre Courbot ; *Spécialisation tardive de systèmes Java embarqués pour petits objets portables et sécurisés* ; thèse de doctorat ; Université des Sciences et Technologies de Lille ; 20 septembre 2006.
- [22] Ouaar Hanane ; *Réseaux de Petri et simulation de systèmes embarqués* ; rapport d'études ; Université Mohamed Khider, Biskra ; 2008.

[23] Maaza Soufiane ; *les systèmes d'exploitation pour les systèmes embarqués* ; rapport d'études ; Université Mohamed kheider, Biskra ; 2008.

[24] Paul Kocher, Ruby Lee, Gary McGraw, Anand Raghunathan et Srivaths Ravi ; *Security as a New Dimension in Embedded System Design* ; article ; DAC '04 Proceedings of the 41st annual Design Automation Conference, ACM New York, NY, USA ; 2004 ; p. 753-760.

[25] Pham Viet Tan Nguyen ; *Sécurité pour les téléphones portables* ; rapport de stage de fin d'études ; Ecole Nationale Supérieure des Télécommunications, Paris ; janvier 2005.

[26] Mahmoud Chaira ; *Cartes à puces* ; rapport d'études ; Université De Laghouat, École Doctorale STIC, Option : IRM ; 2010.

[27] Wikipédia ; *Carte_à_puce*.

http://fr.wikipedia.org/wiki/Carte_à_puce ; 2011

[28] Baptiste Besson, Magid Aberkane ; *Les Java Cards* ; rapport technique ; Université de Nice-Sophia Antipolis ; 14 juin 2004.

[29] Sun Microsystems, Inc. ; *Java Card Applet Developer's Guide* ; 17 Juillet 1998.

<http://java.sun.com/products/javacard>

[30] Surender Reddy Adavalli ; *Smart Card Solution: Highly secured Java Card Technology* ; rapport technique ; Department of Computer Science, University of Auckland ; 2002.

[31] Zhiquan Chen ; *Technology for Smart Cards: Architecture and Programmer's Guide* ; livre ; Published by Addison Wesley ; Juin 2000 ; p. 29-48.

<http://java.sun.com/developer/Books/consumerproducts/javacard/ch03.pdf>

[32] Michel Dufner ; *Les Java Card* ; rapport de fin d'études ; ENSEM ; 2008.

[33] Serge Chaumette et Jonathan Ouoba ; *Java Card U(SIM) et applications sécurisées sur téléphones mobiles* ; article ; Misc (Multi-system & Internet Security Cookbook) Hors-Série N° 2 ; Novembre 2008.

[34] Christophe Bidan et Pierre Girard ; *La sécurité des cartes à microprocesseur* ; article ; Publié dans Revue de l'Electricité et de l'Electronique (REE), n°5 ; Mai 2001 ; p. 60-65.

[35] Marc Witteman ; *Java Card Security* ; article ; Information Security Bulletin, volume 8 ; Octobre 2003 ; p. 291-298.

- [36] Julien Iguchi-Cartigny, Jean-Louis Lanet ; *Evaluation de l'injection de code malicieux dans une Java Card* ; article ; Invited Conference, SSTIC 09, Rennes ; Juin 2009.
- [37] Wojciech Mostowski et Erik Poll ; *Malicious Code on Java Card Smartcards : Attacks and Countermeasures* ; article ; Proceedings, Smart Card Research and Advanced Application Conference CARDIS, Egham, U.K.; Septembre 2008 ; p. 1-16.
- [38] Vertanen, O. ; *Java Type Confusion and Fault Attacks* ; Lecture Notes in Computer Science, vol. 4326/2006, Springer Berlin, Heildeberg ; 2006 ; p. 237-251.
- [39] Hypponen, K. ; *Use of cryptographic codes for byte code verification in smart card environment* ; mémoire de master, University of Kuopio ; 2003.
- [40] Ludovic Casset ; *Construction Correcte de Logiciels pour Carte à Puce* ; thèse de doctorat ; Université d'Aix-Marseille II et Université de la Méditerranée, Ecole Doctorale de Mathématiques et Informatique ; Octobre 2002.
- [41] T. Lindholm et F. Yellin ; *The Java Virtual Machine Specification* ; livre ; Addison-Wesley, Second edition ; 1999.
- [42] Mathieu Corbeil ; *Vérification de code-octet avec sous-routines par code-certifié* ; mémoire présenté comme exigence partielle de la maîtrise en informatique ; Université du Québec à Montréal ; Juillet 2007.
- [43] Jamal Lazaar ; *Preuve de validité du vérificateur de code octet Java* ; mémoire présenté comme exigence partielle de la maîtrise en informatique ; Université du Québec à Montréal ; Décembre 2008.
- [44] Damien Deville ; *Développement d'un vérifieur de bytecode JavaCard optimisé pour carte à microprocesseur* ; mémoire de DEA ; Laboratoire d'Informatique Fondamentale de Lille ; 2 Juillet 2001.
- [45] Leroy X. ; *On-Card Byte Code Verification for Java Card* ; article ; Proceedings of e-Smart 2001 , Cannes ; Septembre 2001.
- [46] Necula G., Lee P. ; *Proof-Carrying Code* ; article ; In 24 th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris ; 1997 ; p. 106-119.

- [47] Sun Microsystems, Inc. ; *Java CardTM 2.2 Off-Card Verifier* ; Juin 2002.
<http://java.sun.com/products/javacard>
- [48] Sun Microsystems, Inc. ; *Java Card 2.1.1 Virtual Machine (JCVM) Specification* ; 18 Mai 2000.
<http://java.sun.com/products/javacard>
- [49] Leroy X. ; *Java Byte Code Verification : An Overview* ; article ; Proceedings of Computer Aided Verification, CAV'2001, LNCS 2102, Springer-Verlag ; 2001 ; p. 265-285.
- [50] Sun Microsystems, Inc. ; *Development Kit User's Guide, Java Card Platform, Version 2.2.2* ; Mars 2006.
<http://java.sun.com/products/javacard>
- [51] Sun Microsystems, Inc. ; *Java Card 2.2 Virtual Machine Specification* ; Juin 2002.
<http://java.sun.com/products/javacard>
- [52] Sun Microsystems, Inc. ; *Java Card 2.2.2 Application Programming Interface Specification* ; Mars 2006.
<http://java.sun.com/products/javacard>
- [53] Sun Microsystems, Inc. ; *Java Card 2.2.2 Runtime Environment Specification* ; Mars 2006.
<http://java.sun.com/products/javacard>
- [54] Julien Iguchi-Cartigny, Jean-Louis Lanet ; *Developing a Trojan applets in a smart card* ; article ; Journal in Computer Virology ; Septembre 2009.
- [55] Borland ; *Introduction à Java* ; livre ; Borland JBuilderTM , version 3 ; 1999.
<http://Rangiroa.essi.fr/cours/langage/99-java-intro.pdf>
- [56] Souheib Baarir ; *Cours JAVA : Introduction à Java* ; rapport technique ; Université Paris Ouest Nanterre La Défense, Laboratoire d'informatique de Paris 6 ; 2010.
- [57] Wikipédia ; *Plate-forme JAVA*.
http://fr.wikipedia.org/wiki/Plate-forme_JAVA ; 2011.
- [58] S. Laporte ; *Introduction à Java* ; cours de développement ;
<http://stephanie.laporte.pagesperso-orange.fr/Pdf/INTRODUCTION%20A%20JAVA.pdf> ; 2011.

Annexe :
Le concept Java

1. INTRODUCTION

Il est important de clarifier ce que représente exactement le mot Java. Java est plus qu'un simple langage informatique ; c'est un environnement informatique. En effet, Java représente deux composants inséparables : Java de conception (le langage lui-même) et Java d'exécution (la plate-forme).

La plate-forme et le langage Java sont issus d'un projet de Sun Microsystems datant de 1990. Généralement, on attribut sa paternité à trois de ses ingénieurs : James Gosling, Patrick Naughton et Mike Sheridan. Il a été créé avec l'objectif de pouvoir exécuter les programmes sans recompilation sur n'importe quelle machine et il est devenu aujourd'hui l'un des langages de programmation les plus utilisés. Il est incontournable dans plusieurs domaines : Systèmes dynamiques (chargement dynamique de classes), Internet (Les Applets Java), les systèmes communicants (RMI, Corba, EJB), etc. [55] [56]

La possibilité d'exécuter les programmes Java dans plusieurs machines fait de Java un langage portable et idéal pour les systèmes mobiles et les applications web (applets, services web). Ceci conditionne ces environnements à exécuter le code octet en ignorant sa provenance et sa source ; par conséquent, ils ne peuvent ni savoir ce qu'il peut réaliser, ni ses vrais effets avant qu'il soit exécuté. Il est donc, essentiel d'assurer le bon fonctionnement des systèmes qui utilisent Java, leur intégrité, leur confidentialité et leur sécurité. Le modèle de sécurité de Java repose sur la définition du langage lui-même (par exemple, l'impossibilité de manipuler directement les pointeurs) et sur certains composants de la plate-forme (machine virtuelle) qui utilisent des mécanismes spécifiques pour forcer certaines restrictions de sécurité sur les programmes Java (par exemple, le gestionnaire de sécurité détermine les droits d'accès d'un programme). [43]

Dans cette partie d'annexe, nous allons présenter le langage et la plate-forme Java, les formes que peut prendre un programme Java, ainsi que son cycle de vie depuis sa création jusqu'à son exécution. En suite, nous allons présenter l'aspect sécurité dans Java et les différents fonctionnalités qu'offre le langage et la plate-forme pour gérer la sécurité des programmes.

2. LE LANGAGE JAVA

Le langage Java est un langage de programmation informatique orienté objet. On définit un objet comme un modèle de programmation, il a un état et un comportement. Dans l'implémentation d'un objet, son état est défini par ses variables d'instance, elles sont propres à l'objet. Le comportement d'un objet est défini par ses méthodes. La classe est une structure qui définit les variables d'instance et les méthodes d'un objet.

Java est un langage hybride, à la fois compilé et interprété. On dit qu'il est semi-compilé. Pour simplifier, disons qu'un programme Java est compilé dans un langage qui devra ensuite être interprété. Le résultat de la compilation n'est pas du langage machine directement exécutable (propre au processeur), mais un code intermédiaire appelé bytecode. Le bytecode est intermédiaire entre le code source et le langage machine.

Pour exécuter le programme, le bytecode est interprété par un interpréteur appelé **machine virtuelle Java** (JVM). Toutes les machines actuelles possèdent une JVM. Ainsi, le bytecode d'un programme peut être exécuté sur n'importe quel ordinateur (possédant une JVM) alors qu'un programme compilé en langage machine n'est exécutable que sur un seul type de processeur. C'est pour cela que le langage Java est un **langage portable**.

La syntaxe de Java se fonde très fortement sur le langage C++, dont il est par ailleurs un sous-ensemble par certains aspects, alors que d'autres caractéristiques du langage sont réellement originales. C++ est un langage à objets, Java ne reprend pas tous les aspects de ce langage, certains choix ont été faits. Voici quelques exemples :

- La notion de pointeur (dans C++) n'existe pas en Java, il n'y a que la notion de référence.
- Il n'y a plus de destructions explicites des objets, tout est géré par un ramasse-miettes (*garbage collector*). Ce ramasse-miettes détruit les objets qui ne sont plus référencés.
- Une nouvelle notion apparaît dans Java : la notion de *package*. Un *package* permet de regrouper plusieurs définitions de classes ou d'interfaces. [1]

3. LA PLATE-FORME JAVA

La plate-forme Java est une plate-forme produite par Sun Microsystems permettant de développer et d'exécuter des programmes écrits en langage Java indépendants de tout processeur et de tout système d'exploitation. Elle est constituée de plusieurs programmes, chacun d'entre eux apportant une fonctionnalité de l'ensemble de ces capacités.

La plate-forme Java se compose principalement d'un JRE (Java Runtime Environment) qui désigne un ensemble d'outils permettant l'exécution de programmes Java sur toutes les plates-formes supportées. JRE est constitué d'une JVM (Machine Virtuelle Java), le programme qui interprète le code Java compilé (bytecode) et le convertit en code natif. Mais le JRE est surtout constitué d'un ensemble de bibliothèques standards dont il existe plusieurs implémentations pour divers matériel et système d'exploitation, de façon à ce que les programmes Java puissent s'exécuter de façon identique sur chacun d'entre eux. Un composant essentiel de la plate-forme Java est le compilateur Java qui permet de convertir les codes source Java en bytecode Java, il est fourni par le JDK (Java Development Kit).

▪ Remarque

Le JRE est à distinguer du JDK (Java Development Kit) qui est constitué en plus d'outils de développement permettant essentiellement de compiler du code Java pour produire du bytecode qui sera interprété par la machine virtuelle sur le poste utilisateur.

Il existe plusieurs versions de plates-formes Java, on retrouve notamment :

- La *Java Standard Edition* (ou Java SE, et auparavant J2SE) destinés aux ordinateurs de bureau ;
- La *Java Enterprise Edition* (ou Java EE, et auparavant J2EE) destinés aux serveurs Web ;
- La *Java Micro Edition* (ou Java ME, et auparavant J2ME), destinés aux appareils portables comme les smartphones. [57]

3.1. La machine virtuelle Java

La machine virtuelle Java (Java Virtual Machine ou JVM) est l'environnement d'exécution des programmes Java. Elle définit principalement un ordinateur abstrait et précise les instructions que ce dernier peut exécuter. Ces instructions sont appelées *bytecodes*. De façon générale, le bytecode Java est à la JVM ce que le jeu d'instructions est à une CPU. Lors de la compilation d'un fichier *.java* (le fichier comportant le code source), le compilateur produit une suite de bytecodes qu'il stocke dans un fichier *.class*. L'interpréteur Java peut ensuite exécuter les bytecodes stockés dans le fichier *.class*. [55]

La JVM est spécifique à chaque plate-forme ou couple (machine/système d'exploitation) et permet aux applications Java compilées en bytecode de produire les mêmes résultats quelle que soit la plate-forme, tant que celle-ci est pourvue de la machine virtuelle Java adéquate. [57]

La JVM est responsable de l'exécution des fonctions suivantes :

- Affectation de mémoire aux objets créés.
- Récupération des données périmées (garbage collection).
- Gestion du recensement et des piles.
- Appel du système hôte pour certaines fonctions, comme l'accès aux périphériques.
- Suivi de la sécurité des programmes Java.

Dans cette partie d'annexe, nous allons nous concentrer sur l'aspect sécurité dans Java.

La JVM débute son exécution par le chargement (*loading*) de la classe contenant une méthode appelée *main*, cela en utilisant le chargeur de classes (*bootstrap Loader*), ou bien un chargeur de classes défini par l'utilisateur. Ensuite, elle aborde la deuxième étape qui consiste à faire la liaison des liens qui se déroule elle-même en quatre étapes :

- **La vérification** : Elle consiste en la vérification des contraintes statiques et structurelles et la validation du format du fichier *.class*
- **La préparation** : Elle consiste en la création des champs statiques de la classe, en initialisant leurs valeurs par défaut.
- **La résolution** : Elle permet de transformer les références symboliques qui sont stockées dans le tableau des constantes (*Constantpool*) et qui peuvent causer d'autres chargements de classes.
- **Le contrôle d'accès** : Il consiste en la vérification des permissions et des droits d'accès à d'autres champs, méthodes ou classes.

Une fois que la liaison des liens s'est bien déroulée, l'initialisation prend place dans le processus d'exécution, qui consiste en l'initialisation des champs statiques, et en l'invocation des initialiseurs statiques (séquence d'instructions suivant la déclaration des champs, délimitée par des accolades et précédée par le mot *static*). [43]

3.2. Les APIs de Java

Pour prendre en charge certaines fonctionnalités, la plupart des langages de programmation reposent sur des bibliothèques déjà construites. Le JDK dispose d'une bibliothèque très impressionnante qui offre au programmeur un certain nombre de classes. On les appelle *API (Application Programming Interface)*.

Les APIs fournissent de nombreux packages essentiels au développement Java. Ces packages incluent le langage, les E/S, les utilitaires, le réseau, l'AWT (Abstract Window Toolkit), le texte, la sécurité, l'RMI (Remote Method Invocation), la réflexion et l'SQL (Structured Query Language). Elles rendent transparente l'architecture sur laquelle est susceptible de s'exécuter les programmes. Nous allons présenter ici les 4 bibliothèques les plus utilisées :

- **java.lang** : contient les classes principales de prise en charge du langage. Il est pratiquement impossible d'écrire un programme Java sans utiliser ce paquet.
- **java.io** : contient les classes pour la gestion des entrées-sorties.

- **java.util** : contient les classes et interfaces pour la gestion des dates, de l'heure, des dictionnaires, etc.
- **java.awt** : contient les classes qui gèrent les fonctionnalités nécessaires pour élaborer des interfaces graphiques, comme la gestion des fenêtres, des couleurs, des polices, etc. [1] [55]

4. CYCLE DE VIE D'UN PROGRAMME JAVA

Le développement d'un programme Java nécessite l'installation du JDK (Java Development Kit) qui inclut le JRE et le plugin. Il est possible d'utiliser des environnements de développement comme JBuilder (BorlandInprise), Eclipse (IBM), Netbeans, etc. Mais une grande partie des développeurs n'utilisent qu'un éditeur avec le JDK.

Le fichier source doit être enregistré avec l'extension *.java* (au lieu de *.cpp* en C++) et porter le même nom que la classe qui contient la fonction principale *main*.

La compilation génère des fichiers de bytecode avec l'extension *.class*, chaque classe du fichier donne lieu à la création d'un fichier *.class*.

Ensuite, l'exécution diffère selon la forme du programme Java. Un programme Java peut être une application, une applet ou une servlet.

- Une **application** Java est un programme indépendant du Web, exécuté en local (comme tous les programmes C++), lancé localement sur la machine.
- Une **applet** Java est un programme destiné à être inséré dans une page Web. Une applet est exécutée par le client (comme les programmes). C'est le fichier compilé en bytecode qui est envoyé au client (et non le fichier source comme dans le cas de javascript). Dans une page Web, une applet est considérée comme une image.
- Une **servlet** Java est un programme exécuté sur le serveur, qui permet de répondre à une requête d'un client http. Une servlet reçoit une requête http avec des données (paramètres), elle traite ses données (souvent accès à une BD) et la plupart du temps elle renvoie une réponse sous la forme d'une page web dynamique. [58]

Avant de décrire comment se déroule l'exécution des programmes Java, il est nécessaire de présenter d'abord leurs structures.

4.1. Structure d'un programme Java

Un programme Java est constitué d'un ensemble de classes. Aucune partie de code ne peut être écrite en dehors d'une classe. Cela vient du fait que, contrairement au C++, Java est un PUR langage orienté-objet. Même la fonction *main*, point d'entrée des applications Java est incluse dans une classe.

Le fichier *.java* doit obligatoirement porter le nom de la classe (ou s'il en a plusieurs, le nom de celle qui est publique).

Une **application** Java doit toujours posséder une classe contenant une méthode appelée *main*. Cette fonction est celle qui est exécutée en premier et qui appelle toutes les autres (c'est le programme principal en quelque sorte). Plus précisément elle commence ainsi :

```
public static void main (String args[ ])
```

args est un tableau de chaînes qu'on peut passer en paramètre du programme principal dans la commande d'exécution.

- **Exemple d'une application Java**

Cette application se contente d'afficher Bonjour à l'écran :

```
public class Bonjour //le fichier doit s'appeler Bonjour.java
{
public static void main (String args[ ]) //fonction principale
{
System.out.println("Bonjour");
}}
```

Une **applet** quand à elle doit contenir une fonction appelée *paint*, chargée de dessiner l'image du résultat de l'applet dans la page Web. Une applet ne doit pas contenir de fonction *main*. C'est la fonction *paint* qui est cherchée en premier par la JVM intégrée au navigateur et pas la fonction *main*.

- **Exemple d'une applet Java**

```
// importation des packages (ensembles de classes) nécessaires
pour écrire des applets
import java.awt.Graphics;
import java.applet.Applet;
public class Bonjour extends Applet { //Le fichier doit
//s'appeler Bonjour.java
public void paint(Graphics g) { // fonction principale
// pour une applet
g.drawString("Bonjour !", 10, 10); // chaque affichage se fait
// dans une fenêtre avec une largeur et une hauteur en pixels
}}
```

Cette applet, pour être exécutée, devra être insérée dans une page Web qui pourrait ressembler à ceci :

```
<HTML>
<HEAD>
<TITLE>Page HTML support de l'applet Bonjour</TITLE>
</HEAD>
<BODY>
<APPLET CODE="Bonjour.class" WIDTH=100 HEIGHT=50>
Votre navigateur ne supporte pas les applets ! (ceci est le
message qui s'affiche par défaut dans le cas où le navigateur
ne sait pas exécuter les applets)
</APPLET>
</BODY>
</HTML>
```

4.2. Exécution d'une application

L'exécution d'une application Java nécessite l'installation du JRE (Java Runtime Environment) sur la machine de l'utilisateur. L'application est exécutée par la machine virtuelle Java à partir de fichiers (de type *.class*) de bytecode (les fichiers peuvent être regroupés en une archive de type *.jar*).

4.3. Exécution d'une applet

L'exécution d'une applet Java nécessite l'installation du JRE (Java Runtime Environment) sur la machine de l'utilisateur et du plugin Java utilisant le JRE pour les navigateurs.

Pour exécuter une applet, il faut évidemment qu'elle soit intégrée dans une page Web. L'exécution de l'applet se fait automatiquement après chargement de la page Web. Ce qui est intégré dans la page, c'est le nom du fichier semi-compilé *.class*. Au moment du chargement de la page, le fichier semi-compilé est envoyé au client, et c'est la JVM du navigateur qui exécute le bytecode. L'applet est donc exécutée par le client et non par le serveur (contrairement aux programmes en ASP (Active Server Pages) ou PHP (Hypertext Preprocessor)).

4.4. Exécution d'une servlet

C'est le serveur web, recevant une requête http, qui va lancer automatiquement la servlet chargée de traiter la requête http. On peut dire que c'est l'internaute qui lance la servlet. [58]

5. LE MODELE DE SECURITE DE JAVA

Dans cette section, la distinction entre une application et une applet est superflue. Nous utiliserons dans ce cas le mot *app* pour faire référence aux applications Java comme aux applets Java.

La possibilité d'exécuter les apps Java dans plusieurs machines conditionne ces environnements à exécuter le code octet en ignorant sa provenance et sa source; par conséquent, ils ne peuvent ni savoir ce qu'il peut réaliser, ni ses vrais effets avant qu'il soit exécuté. Il est donc, essentiel d'assurer la sécurité des apps Java. Le modèle de sécurité de Java repose sur la définition du langage lui-même et sur certains composants de la machine virtuelle qui utilisent des mécanismes spécifiques pour forcer certaines restrictions de sécurité sur les apps Java. [43]

Ce mécanisme (ou modèle de sécurité) joue les rôles suivants :

- Il vérifie que tout fichier téléchargé l'est à partir d'une source certifiée.
- Il contrôle que les bytecodes n'effectuent pas d'opérations interdites.
- Il vérifie que chaque bytecode est généré correctement.

Nous allons présenter ici, les composants de la JVM qui sécurisent les apps Java. En particulier, nous allons examiner les rôles du vérifieur Java, du gestionnaire de sécurité et du chargeur de classe. Nous verrons en outre comment les spécifications du langage Java sont un facteur important pour faire de Java un environnement sûr.

5.1. Le vérifieur Java

A chaque chargement d'une classe, il faut commencer par une vérification. Le rôle principal de cette vérification consiste à s'assurer que chaque bytecode de la classe ne viole pas les spécifications de la machine virtuelle Java. Par exemple, des erreurs de syntaxe et des opérations arithmétiques en débordement ou en sous-débordement sont des violations. La vérification est effectuée par le *vérifieur Java* et se décompose en quatre étapes :

La première étape du vérifieur concerne le contrôle de la structure du fichier class. Tous les fichiers class partagent une structure commune.

La deuxième étape exécute des vérifications au niveau système : validité de toutes les références à la réserve de constante et assurance que toutes les sous-classes sont correctes.

La troisième étape valide les bytecodes. C'est l'étape du processus de vérification la plus importante et la plus complexe. Valider un bytecode signifie contrôler la validité de son type ainsi que le nombre et le type de ses arguments. Le vérifieur contrôle aussi que les appels aux méthodes transmettent des arguments dont le nombre et le type sont corrects et que chaque fonction externe renvoie le type exact. Pour terminer, le vérifieur s'assure de la bonne initialisation de toutes les variables.

L'étape finale effectue les contrôles d'exécution. Les classes référencées en externe sont chargées et leurs méthodes sont contrôlées.

Il est important de noter que le processus de vérification doit avoir lieu au niveau du vérifieur et non à celui du compilateur, puisque n'importe quel compilateur peut être programmé pour générer des bytecodes Java. Il est donc clair que se fier au compilateur pour exécuter la vérification est dangereux, puisque le compilateur peut être programmé pour ne pas en tenir compte.

5.2. Le gestionnaire de sécurité

La classe *SecurityManager* est une des classes définies dans le package `java.lang`. Elle définit la stratégie de sécurité à appliquer aux apps Java. Le rôle principal de cette stratégie consiste à déterminer les droits d'accès. Voici un aperçu de son fonctionnement : chaque app Java chargée dans la JVM existe dans son propre *espacenom*. L'*espacenom* d'une app définit ses limites d'accès. Cela signifie que l'app ne peut pas accéder aux ressources situées au-delà de son *espacenom*.

Avant qu'une app puisse accéder à une ressource du système, comme un fichier local ou sur réseau, l'objet *SecurityManager* vérifie que cette ressource se trouve à l'intérieur de l'*espacenom* de l'app. Dans ce cas, l'objet *SecurityManager* accorde les droits d'accès ; sinon, il les refuse.

5.3. Le chargeur de classe

Le chargeur de classe s'associe au gestionnaire de sécurité pour gérer la sécurité des apps Java. Les rôles principaux du chargeur de classe sont résumés ci-dessous :

- Il charge des fichiers classe dans la machine virtuelle.
- Il identifie le paquet auquel une classe chargée appartient.
- Il trouve et charge toutes les classes référencées par la classe actuellement chargée.
- Il vérifie les tentatives d'accès effectuées par la classe chargée à des classes situées en dehors de son package.

- Il garde une trace des sources des classes chargées et vérifie que les classes sont chargées à partir de sources valides.
- Il fournit au gestionnaire de sécurité certaines informations relatives aux classes chargées.

5.4. Sécurité du langage

Java lui-même est un langage sûr. Il décourage des programmes malveillants et fournit donc l'intégrité des données et la sécurité. Voici quelques bases de la sécurité du langage Java :

- Le langage Java est fortement typé. Ainsi on ne peut pas convertir des entiers en pointeurs.
- Les variables doivent être initialisées avant d'être utilisées.
- Le niveau d'accès de toutes les classes, méthodes et champs est strictement contrôlé par les modificateurs d'accès. Par exemple, une méthode peut être déclarée comme étant publique, protégée ou privée. Une méthode privée ne peut pas être invoquée à l'extérieur de sa classe.
- Le langage Java n'a pas de pointeur arithmétique. Ainsi on ne peut pas laisser les pointeurs introduire des virus à l'intérieur de la mémoire. [5]

En outre, certaines APIs de Java permettent d'effectuer des opérations cryptographiques, comme le chiffrement, le déchiffrement et les signatures. Ces opérations peuvent être utilisées lors du stockage ou de l'échange d'informations sensibles pour garantir un certain niveau de sécurité. [1]

6. CONCLUSION

Java a gagné beaucoup de popularité dans l'industrie dans les deux dernières décennies. Les principales raisons de son succès sont qu'il permet aux applications d'être développées dans un langage de haut niveau indépendamment de tout matériel et qu'il dispose des mécanismes de sécurité qui garantissent l'innocuité des applications téléchargées.

Aujourd'hui, la plate-forme Java s'est imposée dans plusieurs systèmes. Parmi ces systèmes, on trouve les cartes à puce, particulièrement, les Java Cards, qui sont des cartes à puce qui utilisent un langage hérité de Java et adapté aux petits appareils.

Dans cette partie d'annexe, nous avons présenté quelques concepts de Java qui vont servir de bases pour la définition des Java Cards.

ملخص

البطاقات الذكية أصبحت الآن أجهزة كمبيوتر مصغرة و مبرمجة بلغات كائنية التوجه. قدراتهم الحسابية و التخزينية تسمح لهم باحتواء العديد من التطبيقات. قد تم اليوم، تجهيز الكثير من هذه البطاقات بمجموعة فرعية من منصة جافا (Java). و يطلق على هذه البطاقات اسم بطاقة جافا.

استخدام جافا في مجال البطاقات الذكية يعطي هذه البطاقات القدرة على تحميل برامج قابلة للتنفيذ بعد صدور البطاقة. يتم تحميل البرنامج على البطاقة في شكل محوّل (بايت كود). بعد تحويل هذا البرنامج، يمكن أن يعبر من خلال شبكة للوصول إلى البطاقة. لا شيء يضمن أنه لم يتم تغيير في هذا البرنامج وأنه لا يحتوي على هجوم يمكن أن يهدد أمن البطاقة الذكية كمنصة للتنفيذ.

الآلية الأمنية الرئيسية التي تستخدمها بطاقة جافا للتعامل مع مثل هذه الهجمات هي استعمال متحقق البايت كود الذي توفره شركة SUN Microsystems. في هذه المذكرة، نقترح ونصنع وحدة تحقق من البايت كود و التي يمكنها اكتشاف أخطاء خلط النوع التي لا يكتشفها متحقق البايت كود الموفر من SUN، ونعيد إنتاج متحقق جديد وذلك بإضافة وحدة التحقق التي صنعناها إلى متحقق البايت كود الموفر من SUN. الكلمات الرئيسية: الأنظمة المدمجة، البطاقات الذكية، بطاقة جافا، الأمن، التحقق، بايت كود.

Résumé

Les cartes à puce sont devenues maintenant des mini-micro-ordinateurs programmables avec des langages orientés objet. Leurs capacités de calcul et de stockage leur permettent de contenir plusieurs applications. Aujourd'hui, une grande quantité de ces cartes sont équipées avec un sous-ensemble de la plate-forme Java. Elles sont appelées Java Cards.

L'utilisation de Java dans les cartes à puce leur apporte la possibilité de charger du code exécutable après la délivrance de la carte. Ce code est chargé sur la carte sous forme de code intermédiaire (bytecode). Après la compilation de ce code, il peut transiter sur un réseau pour arriver à la carte. Rien n'assure que ce code n'a pas été modifié et qu'il ne contient pas une attaque qui peut menacer la sécurité de la carte à puce en tant que plate-forme d'exécution.

Le point clé des mécanismes de sécurité mises en œuvre par Java Card pour faire face à ce type d'attaques est l'utilisation du vérifieur de bytecode fournis par SUN Microsystems. Dans ce mémoire, nous proposons et réalisons un module de vérification de bytecode qui permet de détecter des erreurs de confusion de type qui ne sont pas détectées par le vérifieur de bytecode de SUN. Nous reproduisons un nouveau vérifieur en ajoutons le module de vérification que nous avons réalisé au vérifieur de bytecode de SUN.

Mots clés : Systèmes Embarqués, Cartes à puce, Java Card, Sécurité, Vérification, Bytecode.