UNIVERSITÉ DE LIMOGES
ÉCOLE DOCTORALE « Sciences et Ingénierie pour l'Information »
FACULTÉ DES SCIENCES ET TECHNIQUES

Thèse N° 69/2012

THÈSE

pour obtenir le titre de

DOCTEUR DE L'UNIVERSITÉ DE LIMOGES

Discipline / Spécialité : Informatique

présentée et soutenue par

Nassima Kamel

le 20 décembre 2012

Sécurité des cartes à puce à serveur Web embarqué

Thèse dirigée par Jean-Louis Lanet,
co-encadrée par Julien Iguchi-Cartigny

JURY

Rapporteurs :

Mme. Samia Bouzefrane	Maître de Conférences (HDR) au CNAM de Paris
M. Christophe Bidan	Maître de Conférences (HDR) à Supélec de Rennes

Examinateurs:

Mme. Marie-Laure Potet	Professeur à Ensimag, Grenoble INP
M. Christophe Clavier	Professeur à l'Université de Limoges
M. Jean-Louis Lanet	Professeur à l'Université de Limoges
M. Julien Iguchi-Cartigny	Maître de Conférences à l'Université de Limoges

Remerciements

Je tiens tout d'abord à remercier les rapporteurs Dr. Samia Bouzeframe et Dr. Christophe Bidan, d'avoir accépté de faire un compte rendu de ce mémoire et pour les corrections qu'ils ont apportées. Je remercie également les examinateurs Pr. Marie-Lore Potet et Pr. Christophe Clavier d'avoir bien voulu consacrer de leur temps à juger ce travail.

Je remercie mon directeur de thèse Pr. Jean-Louis Lanet de m'avoir donné l'opportunité d'atteindre cet objectif de préparer une thèse doctorale en me proposant ce sujet, et de m'avoir dirigée tout le long de ces trois années.

Je remercie également Dr. Julien Iguchi-Cartigny mon codirecteur, notament d'avoir contribué la définition de mon sujet de thèse.

Mes remerciements vont également à mes adorables collègues (doctorants et ingénieurs), je cite : Amaury, Bhagya, Damien, David, Guillaume et Pierrick, pour leur amitié et pour tous les bon moments de rire et de folie que nous avons passés ensemble. Je remercie encore une fois ceux d'entre eux qui ont eu la gentillesse de lire des parties de ce mémoire.

Je remercie aussi tous les autres doctorants : Aymrick, Antoine, Richard, Slim, Julien ; les anciens doctorants : Oana, Alex, Agnès, Ahmadou et d'autres ; les stagiaires passés et présents de notre équipe ainsi que le personnel technique et administratif en particulier Hubert, Sylvie et Odile, pour leurs amitié et sympathie.

Je n'oublie pas aussi de remercier mes amies de longue date, Fatiha, Kahina et Samira pour leur présence.

Mes plus vifs remerciement vont à ma chère famille qui inclue mon fiancé, de m'avoir soutenue et encouragée dans les moments difficiles. À mes parents particulièrement, je dédie ce titre de DOCTEUR dont ils seront sans doute plus heureux que je ne le suis.

Merci à tous ceux qui ont contribué de prêt ou de loin à l'aboutissement de cette thèse.

Résumé

Les cartes à puces sont des dispositifs sécurisés, de plus en plus utilisés dans le monde. Leur succès est principalement dû à leur aspect *tamper-resistant* qui permet de stocker des informations sensibles (clés de chiffrement) de manière sécurisée. Vu les nombreux sensibles domaines d'utilisation des cartes à puce (bancaire, médical, téléphonie), plusieurs recherches se sont intéressés à la sécurité et aux attaques possibles sur ces dispositifs.

La nouvelle génération de cartes à puce définit un serveur Web embarqué. Il existe principalement deux types de spécifications pour ces nouveaux dispositifs, la première a été établie par l'organisation OMA qui propose un simple serveur HTTP nommé *Smart Card Web Serveur* (SCWS) indépendant de la plateforme d'exécution. La seconde est une nouvelle version de Java Card proposée par *Sun Microsystems* (actuellement Oracle), nommée Java Card 3, édition connectée ; elle inclue le support de l'API Java servlet 2.4, et une amélioration significativement de l'API Java Card (thread, String, etc.) et des fonctionnalités de sécurité supportées.

L'intégration du serveur Web dans la carte à puce apporte un ensemble d'avantages et définit un nouvel usage de la carte à puce. En effet, en plus de l'aspect authentification robuste que la carte peut désormais fournir sur le réseau, l'utilisation des standards du Web permet d'avoir une expérience utilisateur continue et enrichit l'aspect et la convivialité des interfaces fournies via un navigateur Web. Face à ces avantages, les risques d'attaques augmentent. En plus des attaques classiques connues sur les cartes à puce (physiques et logiques), certaines attaques Web connues sur des applications Web standards peuvent se reproduire sur les cartes à puce. Parmi ces attaques, la *Cross Site Scripting* (appelée aussi XSS) est l'une des plus répandues ; elle est simple à réaliser mais ses conséquences peuvent être très graves. Cette attaque exploite des applications vulnérables qui ne filtrent pas les données non fiables (entrée par un utilisateur) avant de les utiliser. Elle consiste à injecter un code malicieux dans une entrée fournie par l'application, qui est ensuite retourné dans une ressource de l'application et exécuté dans le navigateur Web de la victime.

D'autre part, les spécifications des cartes à serveur Web embarqué définissent des protocoles (HTTP, BIP, TCP/IP) qui doivent nécessairement être implémentés pour assurer la communication de la carte avec le navigateur Web et sur le réseau. Des failles d'implémentation de ces protocoles peuvent engendrer des vulnérabilités facilitant les attaques sur les cartes à puce.

Dans cette thèse nous nous intéressons à la sécurité des cartes à puce à serveur Web embarqué à deux niveaux. Le premier concerne la sécurité des applications Web contre des attaques XSS. Nous proposons dans ce cadre un outil d'analyse statique des applications Web Java Card 3, qui a pour objectif de vérifier qu'une application est sécurisée contre ces attaques, incluant un filtrage des données non fiables. Notre outil peut être considéré comme un outil de certification de la robustesse des applications Web avant leur chargement dans la carte. Nous avons également implémenté une API de filtrage que le développeur peut importer dans son application.

Le second niveau de sécurité exploré, concerne l'implémentation du protocole HTTP ; nous suggérons un outil de fuzzing pour tester la conformité et la robustesse du protocole HTTP implémenté dans une carte. Cet outil apporte un ensemble d'optimisations qui réduit le temps du *fuzzing* et génère des données de test de manière intelligente.

Mots clé : *carte à puce*, *Java Card* , *SCWS*, *XSS*, *HTTP*, *analyse statique*, *dépendance causale*, *Fuzzing*.

Abstract

Smart cards are widely used secure devices in today's world, which can store data in a secured manner and ensure data security during transactions. The success of smart card is mainly due to their tamper-resistant nature which allows them to store sensitive data's like cryptographic keys. Since they are using in multiple secure domains, like banking, health insurance, etc. more and more researches are taken place in this domain for security and attacks.

The last generation of smart card, defines an embedded web server. There are two types of specifications for these devices, the first one is defined by OMA organisation that propose a simple HTTP web server named Smart Card Web Server (SCWS), the second is proposed by Sun Microsystems (currently Oracle), consists of a Java card 3 connected edition platform, that includes a Java servlet 2.4 API with improved Java Card API and security features. In addition to network benefits from the robustness of smart card, the use of web standards provide a continuous user experience, equivalent to that seen while surfing on the internet and it enhances the look and feel of GUI interfaces. The GUI interfaces are accessible from a browser which is located on the terminal on which the card is connected.

However, in addition to the classical attacks (physical and logical), the integration of web server on smart card, exposes the smart card to some existing classical web application attacks. The most important one is the cross site scripting attack, also named XSS. It consists of injecting malicious data to the given web application inputs and if the resource returned to the browser includes the malicious code, it will be interpreted and executed, causing an attack. A web application is vulnerable to XSS if it uses an untrusted data without filtering malicious characters before. On the other hand, to ensure the communication between web applications and browser or other network entities, it is necessary to integrate some protocols to the smart card, for example HTTP, BIP or TCP/IP. The vulnerabilities in the implementation of these protocols can facilitate some attacks.

Our contribution on this thesis is divided in two parts, in the first part, we are interested on the security of web applications against XSS attack. We suggest a static analysis tool, based on tainting approach, that allow to verify if a web application is secured or not, including filtering data in all insertion points where XSS is possible. We also implement, an API filter, compatible with Java Card 3 platform, that developers can import during the development of their applications. The second part consists of verifying the conformance and the robustness of the implemented HTTP protocol. For that we propose an intelligent fuzzing tool that includes a set of optimisations that allows to reduce the time of fuzzing.

Key words : *smart card*, *Java Card 3*, *SCWS*, *HTTP*, *XSS attacks*, *Fuzzing*, *Static analysis*, *Tainting*.

Sommaire

Introduction générale **1**

I Domaine d'étude **7**

1 Contexte d'étude **9**

1.1 La carte à puce 9

1.2 Historique 10

1.3 Les protocoles de communication 11

1.3.1 Le protocole APDU 12

1.3.2 Le protocole BIP 12

1.3.3 Le protocole HTTP 13

1.4 Les cartes à puce à serveur Web embarqué 13

1.4.1 Les cartes à serveur Web SCWS 14

1.4.2 Les cartes Java Card 3 Edition Connectée 17

1.5 Cas d'utilisation des cartes à puce à serveur Web embarqué 21

1.5.1 Accès rapide et simple aux services offerts par la carte 21

1.5.2 Amélioration de l'interface d'accès aux services offerts par l'émetteur 22

1.5.3 Sécurité de connexion d'un terminal distant à une application 22

1.5.4 Exemple d'application : « le Disque Virtuel » 23

1.6 Conclusion 24

2 Les attaques sur carte à puce **25**

2.1 Attaques sur cartes à puce standards 26

2.1.1 Attaques physiques 26

2.1.2 Attaques logicielles 27

2.1.3 Attaques combinées 30

2.2 Les attaques Web 31

2.2.1 Exemples d'attaques Web les plus répandues 32

2.3 Conclusion 36

II Sécurité des applications Web Java Card 3 39

3 Les attaques Cross Site Scripting 41

3.1 Description 41

3.1.1 Injection volatile 41

3.1.2 Injection persistante 43

3.2 Portée et exploitation des attaques XSS 44

3.3 Solutions existantes 45

3.3.1 Solutions côté client 46

3.3.2 Solutions côté serveur 47

3.4 Conclusion 52

4 Détection de vulnérabilités dans un programme 53

4.1 Introduction 53

4.2 Analyse statique 55

4.2.1 État de l'art des outils d'analyse statique 56

4.2.2 L'analyse statique dans Java Card 58

4.3 La dépendance causale 59

4.3.1 Description 59

4.3.2 Travaux existants 60

4.4 Conclusion 61

5 Analyse statique de bytecode Java Card 3 63

5.1 Présentation générale 64

5.2 Représentation d'une vulnérabilité XSS 64

5.3 Processus d'exécution d'un *bytecode* 65

5.3.1 Zones de données 66

5.3.2 La frame 67

5.3.3 Les instructions de la JCVM 67

5.4 *Findbugs* et ses limites 69

5.5 Analyse inter-procédurale et interclasses 70

5.5.1 Présentation du concept de CFG 70

5.5.2 Analyse inter-méthodes et inter-classes 72

5.5.3 Imprécision de l'analyse insensible au contexte 72

5.6 Analyse des flux de données 73

5.6.1 La dépendance causale 73

5.6.2 Étiquetage des conteneurs 75

5.6.3 Détection des champs persistants 78
5.6.4 Dépendance causale inter-méthodes 78
5.7 API de filtrage XSS pour des applications cartes à puces 79
5.7.1 Filtrage des données insérées dans un élément HTML 80
5.7.2 Filtrage des données insérées dans des attributs HTML 80
5.7.3 Filtrage des données insérées dans un code *JavaScript* 81
5.7.4 Filtrage et validation des données insérées dans les propriétés de style HTML 82
5.7.5 Filtrage des données insérées dans des valeurs de paramètres d'URL 83
5.8 Conclusion 83

6 Implémentation et expérimentations 85
6.1 Présentation de BCEL 85
6.2 Interprétation abstraite de la JCVM 86
6.2.1 Simulation d'une frame 86
6.2.2 Représentation d'un item 88
6.2.3 Plugin XSSDetector 90
6.3 Implémentation de *JCXSSFilter* 91
6.4 Expérimentations 91
6.4.1 Stratégie d'évaluation de l'outil d'analyse statique 92
6.4.2 Temps de l'analyse 93
6.4.3 Détection de vulnérabilités 94
6.4.4 Faiblesse de l'analyse statique 97
6.4.5 Surcoût du filtrage 100
6.5 Conclusion 101

III Sécurité du protocole HTTP 103

7 Le protocole HTTP et les techniques de vérification 105
7.1 Introduction 105
7.2 Les techniques de vérification et de validation de protocoles 106
7.2.1 La vérification formelle 106
7.2.2 Test de logiciels 107
7.3 Le Fuzzing 108
7.3.1 Définition 108
7.3.2 État de l'art des logiciels de *fuzzing* 109
7.3.3 Le *fuzzing* sur carte à puce 110

7.4 Le protocole HTTP 111
7.4.1 Historique 111
7.4.2 Principe de fonctionnement 111
7.4.3 Les requêtes HTTP 111
7.4.4 Les réponses HTTP 113
7.4.5 Les champs d'entête 114
7.4.6 Accès aux ressources protégées 115
7.4.7 Antémémoire 115
7.5 Conclusion 116

8 Fuzzing HTTP **117**
8.1 Présentation générale 117
8.2 L'application *PyHAT* 118
8.2.1 Présentation 118
8.2.2 Fonctionnalités de *PyHAT* 119
8.2.3 Stratégie d'analyse 120
8.3 L'outil *Smart-Fuzz* 123
8.3.1 Présentation 123
8.3.2 Étude des requêtes HTTP 123
8.3.3 Conception d'une BNF (*Backus Normal Form*) HTTP interactive 128
8.3.4 Modélisation des données de test 128
8.3.5 Configuration de Peach pour les cartes à puce SCWS 129
8.3.6 Le parallélisme 130
8.3.7 Fichiers de journalisation 130
8.3.8 Analyse des événements 131
8.4 Conclusion 132

9 Implémentation et résultats expérimentaux de *Smart-Fuzz* et *PyHAT* **133**
9.1 Implémentation de *PyHAT* 133
9.2 Résultats de l'analyse 136
9.3 L'outil *Smart-Fuzz* 136
9.3.1 Les modèles de données 136
9.3.2 Les modèles d'états 138
9.3.3 Les mutations 139
9.3.4 Configuration des tests 139
9.3.5 Configuration du processus d'exécution 141

9.4 Expérimentations et Résultats 141
9.4.1 Temps d'exécution 141
9.4.2 Failles détectées 142
9.5 Conclusion 143

IV Conclusions et perspectives 145

10 Conclusions et perspectives 147
10.1 Problématique 148
10.2 Principales contributions 148
10.3 Perspectives 149
10.3.1 Analyse des applications Web 149
10.3.2 Fuzzing HTTP 151
10.4 Synthèse générale 152

V Annexes 153

A 155
A.1 Représentation du fichier assesment.xml 155
A.2 Fichiers de journalisation de *Smart-Fuzz* 156

B 157
B.1 Exemple de servlet 157
B.2 Descripteurs de déploiement d'une application Web 158

C 161
C.1 Notions Web 161

Liste des tableaux

2.1 Sélection d'OWASP des dix vulnérabilités les plus répandues en 2010 32

3.1 Exemples d'encodages pour outrepasser une liste noire 50

5.1 Exemples de caractères spéciaux et leur encodage en entités HTML 80

6.1 Temps de l'analyse statique 94
6.2 Résultat d'analyse des applications « Disque virtuel » et « JC3CreditDebit » . . . 94
6.3 Résultats d'analyse de l'application *TestPackage* 95

7.1 Versions du protocole HTTP 111
7.2 Les codes de retour d'une réponse HTTP 114

8.1 Stratégie de détection des champs HTTP implémentés 122
8.2 Les requêtes conditionnelles 127

Table des figures

1.1 Carte à puce 10

1.2 Commande APDU 12

1.3 Réponse APDU 12

1.4 Communication entre SCWS et téléphone mobile via le protocole BIP 13

1.5 Architecture du SCWS 15

1.6 SCWS dans une carte Java Card 2.2 16

1.7 Composents d'une application Web Java Card 19

1.8 Exemple d'application Java Card 3 : le Disque Virtuel 23

3.1 Affichage d'une fenêtre de dialogue par un script 42

3.2 Attaque XSS persistante 43

5.1 Automate d'une vulnérabilité XSS 65

5.2 Représentation des structures de la JCVM 65

5.3 Exemple de graphe CFG 71

5.4 Suivi des flux de données inter-méthodes 79

7.1 Principe du *fuzzing* 109

7.2 Communication en HTTP 112

8.1 Vue générale de l'outil de *fuzzing* HTTP 118

8.2 Fonctionnement de l'application *PyHAT* sur une carte SCWS 119

8.3 Fonctionnement de l'application *Smart-Fuzz* (cas des serveurs SCWS) 123

8.4 BNF de l'entête Date 128

8.5 *Fuzzing* sur SCWS 129

8.6 *Fuzzing* parallèle sur plusieurs cartes à puce 130

Introduction générale

1 Introduction

Les capacités des cartes à puce en termes de taille mémoire et de puissance de traitement, permettent aujourd'hui d'intégrer un serveur Web. Les premières spécifications de cartes à puce à serveur Web embarqué ont été définies par l'organisation *Open Mobil Alliance* (OMA) proposant la spécification *Smart Card Web Server* SCWS qui définit l'interface d'un serveur HTTP/1.0 ou HTTP/1.1 embarqué dans la carte à puce [All08a, All08c, All08b]. Cette spécification est destinée aux cartes SIM. Elle permet une connexion locale entre le client HTTP (navigateur Web) du terminal et le serveur Web, ainsi qu'une administration à distance via un protocole sécurisé (TLS) des applications chargées dans le serveur. Elle fournit un ensemble de ressources statiques et dynamiques gérées par des applications Web (xHTML, JavaScript, etc.), et accessibles à l'utilisateur via le client (navigateur Web). Les spécifications ETSI [ETS10b] définissent une API et des bibliothèques disponibles pour la plateforme Java Card 2.2 utiles au développement des applications Web basées sur des servlets et fournissant des méthodes permettant à une servlet de gérer les requêtes/réponses HTTP.

La spécification SCWS n'est pas liée à la Java Card 3 qui définit également un serveur Web embarqué. La spécification de l'OMA est plus simple, dissociée de Java Card et a pour but unique de fournir un « simple » serveur Web embarqué. Elle correspond à des cartes qui n'ont pas les performances suffisantes pour supporter l'API Java card 3 et les protocoles TCP/IP. Pour communiquer avec le terminal basé sur la pile TCP/IP, des protocoles ont été définis dont *Bearer Independent Protocol-BIP*. Une passerelle BIP est nécessaire dans le terminal pour assurer la communication entre les deux parties. La version Java Card 3.0.2 est la spécification la plus récente de la plateforme Java Card de *Sun Microsystems* (aujourd'hui Oracle). Elle est disponible en deux éditions distinctes : classique et connectée.

La première [JC309a, JC309e, JC309d] décrit une évolution légère (API cryptographiques) de la plate-forme Java Card 2.2.2 et cible des marchés à faible valeur ajoutée en privilégiant la compatibilité avec les plates-formes précédentes. Ce marché représente aujourd'hui plus d'un milliard de cartes par an.

La seconde [JC309c, JC309b, JC309f] est conçue pour la nouvelle génération de cartes à puce et s'adresse donc aux marchés les plus haut de gamme, en particulier les cartes SIM les plus puissantes. Avec cette nouvelle technologie, la carte à puce peut communiquer via les protocoles TCP/IP comme tout autre élément d'un réseau informatique. Ainsi, elle est capable de fournir ou d'accéder à des services du réseau.

La principale nouveauté de cette édition est l'intégration de certains standards Web, notamment HTTP ou HTTPS. Elle conserve les principes des précédentes versions (notamment, l'exécution d'applications multiples et une stricte isolation des applications) et permet le déploiement simultané d'applications des versions précédentes (applets) et d'applications Web (à l'aide d'un nouveau modèle de programmation hérité de JEE : les Servlets [Ser03]). Elle apporte également des évolutions majeures au niveau des APIs et des fonctionnalités supportées (support de types String, programmation multi-tâches, vérifieur de bytecode obligatoire, etc.).

Cette nouvelle technologie oblige à repenser toute la sécurité de ces dispositifs par rapport au modèle traditionnel des cartes à puce. L'utilisation de cette technologie rend la carte plus ouverte et plus accessible aux utilisateurs et potentiellement à des attaquants à travers les protocoles réseau tels que HTTP, TCP/IP et BIP. Des vulnérabilités peuvent provenir de failles d'implémentation des protocoles réseau ou d'un développement non sécurisé des applications Web qui devraient respecter une méthodologie de développement sécurisé.

2 Cadre de la thèse

Les travaux présentés dans cette thèse proposent un ensemble de solutions pour répondre aux nouvelles problématiques que posent les cartes à puce à serveur Web embarqué. Nous nous intéressons particulièrement à la sécurité à deux niveaux. Le premier concerne la sécurité des applications Web contre des attaques par injection de code particulièrement le *cross site scripting*, et le second s'intéresse à la vérification de la conformité et la robustesse du protocole HTTP implémenté dans les cartes à puce.

2.1 Sécurité au niveau applicatif

Dans cette partie nous nous intéressons à la détection et à la prévention des attaques Web par injection de code. En effet, si l'application n'est pas soigneusement développée, ne respectant pas une méthodologie de développement sécurisée [OWA07, OWAa], alors des attaques Web standards pourraient se produire sur la carte à puce. La plus répandue de ces attaques est le *cross site scripting* appelée aussi XSS [KGJE09]. Elle consiste à injecter des données contenant un script malicieux dans une entrée d'une application (généralement du JavaScript). Ce code est ensuite exécuté dans le navigateur d'une victime lorsqu'une ressource de l'application intégrant ce code est chargée sans avoir été filtré au préalable. Cette attaque peut être persistante si le code malicieux est sauvegardé dans une ressource permanente dans le serveur, ou volatile si elle est générée uniquement quand l'utilisateur accède à un lien hostile.

L'interprétation du code malicieux peut engendrer le transfert de données (cookie par exemple) vers le domaine d'un pirate, modifier le comportement en local de l'application (dans le navigateur de la victime) ou encore provoquer une indisponibilité de services.

Nous avons d'abord considéré les solutions existantes à différents niveaux. Une première solution est l'intégration de mécanismes de détection de code malicieux au niveau du serveur (pare-feux applicatifs) [SS02, Mod] . Ce mécanisme possède l'avantage d'être indépendant de l'application. Cependant il est difficile de différencier le code malicieux du code JavaScript classique (prévu dans l'application). En général ces services ne détectent que des séquences connues comme hostiles,

ce qui est aisément contournable en réécrivant le code malicieux avec des encodages différents. Les mécanismes de détection d'intrusion IDS [PHP, Gui08] sont une variante de cette approche, ils consistent à vérifier la légalité des flux qui transitent entre le navigateur et le serveur afin d'empêcher que des ressources soient redirigées vers le domaine d'un attaquant. Cependant, comme toutes les solutions dynamiques (analyse pendant l'exécution du code) cette approche engendre des coûts supplémentaires en termes de temps de réponse du serveur et d'occupation d'espace mémoire par l'IDS. De plus, cette approche nécessite généralement une instrumentation du bytecode. Cette solution ne conviendrait pas à des dispositifs à faibles ressources comme les cartes à puce.

Une deuxième solution est un mécanisme à l'intérieur du navigateur autorisant ou non l'exécution du code JavaScript [MOZ]. La décision de bloquer un code JavaScript se base sur la détection de séquences particulières ou l'utilisation de listes blanches et listes noires. Outre le fait que la détection de séquences est contournable, la décision de classer un code comme potentiellement dangereux est de la responsabilité de l'utilisateur. De plus, il n'y aucune garantie que ce mécanisme soit présent ou actif dans un navigateur communiquant avec la carte.

La solution la plus recommandée consiste à prévenir les vulnérabilités XSS depuis la phase de conception des applications Web en renforçant la sécurité par de bonnes pratiques de développement [OWAa]. Le développeur doit considérer toutes données externes à l'application comme non fiables et donc potentiellement malicieuses. Les réponses retournées à l'utilisateur doivent être filtrées si celles-ci sont générées à partir de données malicieuses. Cette solution est difficile à appliquer car elle demande une attention dédiée de la part du programmeur. Une alternative à cette solution consiste à utiliser une API de filtrage "de confiance" (développée par une tierce partie) capable de filtrer efficacement une variable contenant une donnée non fiable. Cependant, nous n'avons aucune garantie que l'application inclue bien une API filtrage, ni si tous les points d'insertion de données dans l'application ont été filtrés. Une solution serait d'utiliser la technique de dépendance causale pour suivre la propagation des données non fiables dans un chemin d'exécution du code et de vérifier si sur ce chemin les fonctions de filtrage sont appelées avant que la ressource non fiable soit retournée au client ou sauvegardées en mémoire persistante.

Une dépendance causale entre deux objets est présente s'il existe un flux d'information direct ou indirect entre eux. Un exemple de technique pour vérifier si une telle relation existe est de marquer avec une étiquette les entrées (les sources) d'un programme et de propager celle-ci à toute variable assignée à partir de ces entrées ou une variable possédant déjà une étiquette (les dérivations). Si les variables étiquetées atteignent des méthodes considérées comme critiques (on parle des états puits) alors le système considère qu'il existe une vulnérabilité potentielle. À l'opposé, si une variable étiquetée atteint une fonction permettant la détection ou le nettoyage des données alors l'étiquette est enlevée pour signifier que la donnée n'est plus considérée comme critique.

Contribution. Notre solution est basée sur une analyse statique appliquant la dépendance causale permettant de suivre la propagation des données non fiables (étiquetées) dans un code et de vérifier qu'elles sont nettoyées par une fonction de filtrage de confiance avant d'être retournées dans une ressource vers le navigateur de l'utilisateur (XSS volatile) ou sauvegardées en mémoire persistante (XSS persistante). L'objectif de cet outil est de certifier qu'une application est sécurisée contre les XSS (volatiles et persistantes) avant son chargement dans la carte à puce.

Notre outil est basé sur le logiciel d'analyse statique *FindBugs* [Fin] qui est capable d'effectuer une analyse des flux de données à partir d'une abstraction de l'exécution du bytecode. Cependant,

ce logiciel a des limites, en particulier, les analyses inter-procédurale et inter-classes ne sont pas prises en compte.

Nous avons donc amélioré cette analyse afin d'étendre *FindBugs* et permettre la propagation des étiquettes entre objets de même méthode ou de méthodes différentes qui appartiennent à une même classe ou de classes différentes. Ainsi, notre solution se compose de deux outils :

- un outil d'analyse statique, nommé *XSSDetector* sous forme de plugin pour *FindBugs*, capable de suivre l'exécution abstraite d'une application afin de vérifier si chaque entrée passe par une fonction de filtrage de confiance avant d'être retournée à l'utilisateur ;
- une API nommée *JCXSSFilter* basée sur l'API ESAPI d'OWASP [ESA], compatible avec la plateforme Java Card 3 permettant de filtrer les données contre les attaques XSS.

Un programmeur décidant d'utiliser nos outils pour développer son application utilise les fonctions de *JCXSSFilter* afin de nettoyer toutes les entrées dans son programme. Puis une fois l'application réalisée, il exécute l'outil d'analyse statique afin de vérifier que toutes les entrées du programme sont bien nettoyées avant d'être sauvegardées en mémoire persistante ou retournées à l'utilisateur. On peut signaler que notre outil peut être étendu pour détecter d'autres cas qui peuvent être gérés par analyse statique.

2.2 Sécurité au niveau du protocole HTTP

La deuxième contribution de nos travaux s'intéresse au protocole HTTP qui doit être nécessairement ajouté à la carte afin d'assurer la communication entre le serveur Web et le navigateur Web. Ce protocole est sans état (chaque requête est traitée indépendamment des autres). Il permet d'effectuer des demandes de ressources sur le serveur, adressées sous forme d'URIs avec des requêtes complétées par des entêtes. Outre le fait qu'il fonctionne en mode texte avec des requêtes très longues, les nombreuses entêtes composant une requête, ont chacune un comportement particulier qu'il faut caractériser.

De nombreux travaux basés sur les méthodes formelles ont été réalisés pour la vérification des protocoles [CMcMW04, MF05, Boc03, Hol91]. Cependant, cette approche permet simplement de vérifier si des propriétés spécifiées sont respectées dans l'implémentation du protocole. D'autre part, la technique de validation par test consiste à utiliser un jeu de tests qui est appliqué au protocole pour vérifier différents critères dont : la conformité à la spécification, la robustesse, la sécurité. La technique de test la plus classiquement utilisée pour l'évaluation des protocoles réseaux est le *fuzzing*. C'est une technique d'injection de données invalides sur une cible. En fonction des réponses du serveur, il est possible de déduire des faiblesses dans la gestion des entrées de la cible qui peuvent être dues au non respect de la spécification ou à des failles de sécurité. Il existe deux grandes classes de générations d'entrées invalides : à partir de modèles de données ou par mutation de séquences [TDM08]. Dans notre cas, nous avons utilisé la première approche en se basant sur le logiciel Peach [Micb].

Le choix de cette approche est motivé par le fait que les fabricants de cartes ne fournissent généralement aucune information sur l'implémentation des applications installées dans les cartes à puce. Nous supposons donc que nous n'avons aucune connaissance sur l'implémentation du protocole HTTP que nous voudrions tester, que ce soit le code source ou la possibilité d'introspecter les instances en cours d'exécution (boite noire). Nous ne pouvons, donc qu'observer les entrées/sorties sans avoir connaissance du comportement interne de la carte.

Les inconvénients de la technique du *fuzzing* résident dans le temps nécessaire pour appliquer tous les jeux de tests possibles sur le système ciblé et aussi dans l'étude des résultats en sortie afin de détecter ses faiblesses. L'approche classique d'un *fuzzing* pour étudier les résultats d'une séquence invalide consiste à vérifier que la cible fonctionne encore correctement après chaque test envoyé (processus encore vivant, machine répondant à certaines commandes...). Mais dans notre cas, n'ayant accès qu'aux réponses à nos requêtes (et l'absence de réponses parfois), notre analyse consiste donc, à comparer les couples requête/réponse aux résultats attendus fournis par un oracle. Cependant, la génération de cet oracle nécessite une caractérisation du protocole HTTP implémenté. La caractérisation serait difficile à réaliser vu que les fonctionnalités HTTP définies dans les spécifications (RFCs) [FGM+99] ne sont pas forcément toutes implémentées. En effet, la spécification HTTP utilise une terminologie qui définit pour chaque fonctionnalité si celle-ci est obligatoire ou non (utilisation des mots « SHOULD », « MAY » et « MUST »). De plus, le protocole implémenté ne respecte pas forcément la norme HTTP : absence de certaines fonctionnalités, réponses négatives ou invalides à certaines requêtes, etc.

Contribution. L'outil de *fuzzing* que nous proposons est composé de deux outils complémentaires *PyHAT* et *Smart-Fuzz* [KBBL11]. L'outil *PyHAT* permet d'avoir une description des fonctionnalités et des caractéristiques de la cible, la plus précise possible. Il se base sur l'analyse des réponses à des requêtes spécifiées en entrée.

La plus grande difficulté est de pouvoir interpréter les résultats retournés, sachant que certaines réponses n'indiquent pas si une erreur s'est produite ou si le serveur Web ne respecte pas les RFCs. Pour chaque méthode, version de HTTP et entêtes, il a fallu trouver un algorithme d'évaluation. Certains cas sont d'ailleurs indécidables et *PyHAT* considère alors par défaut que la carte supporte ce cas.

PyHAT génère donc une grammaire réduite dont certains paramètres sont fixés. Cette grammaire est ensuite exploitée par notre outil de *fuzzing Smart-Fuzz* basé sur Peach pour la génération des requêtes HTTP envoyés à la carte. L'utilisation de cette grammaire réduit considérablement le nombre de données de test en se limitant uniquement à des fonctionnalités implémentées.

Dans *Smart-Fuzz*, nous avons conçu des modèles de données décrivant les requêtes HTTP et les différents champs qui la composent. Ce sont ces modèles que Peach utilise pour générer les données de test qui sont envoyées à la carte. Cependant Peach ne compte pas parmi ses possibilités de configuration le protocole de communication BIP. Nous avons donc développé une interface permettant de gérer cette communication. Nous avons également optimisé le temps d'exécution de notre outil par la configuration d'un *fuzzing* parallèle sur plusieurs cartes. *Smart-Fuzz* est également composé d'une application d'analyse des événements recueillis durant la phase du *fuzzing* et qui se charge de vérifier les codes d'erreurs retournés et la cohérence des entêtes et des réponses par rapport à un oracle prédéfini.

3 Organisation du document

L'ensemble de ce document est organisé en trois parties et contient en tout neuf chapitres. La première partie s'intitule «domaines d'étude», elle est composée de deux chapitres :

- Le chapitre 1 est un rappel sur la définition d'une carte à puce et sur les évolutions majeures qu'elle a connues. Nous présentons également plus en détails les cartes à puce à serveur Web embarqué auxquelles nous nous intéressons dans cette thèse ;
- Le chapitre 2 aborde les attaques sur cartes à puce en général (physiques et logiques) et présente également quelques exemples d'attaques Web susceptibles de se reproduire sur la cartes à puce à serveur web embarqué.

La deuxième partie présente la première problématique que nous avons traitée et qui concerne la sécurité des applications Web dédiées à des cartes à puce (en particulier Java Card 3) contre des attaques XSS. Elle est composée de trois chapitres.

- Le chapitre 3, définit la vulnérabilité XSS et ses conséquences et présente un état de l'art des différentes approches utilisées pour la prévention contre cette vulnérabilité ;
- Le chapitre 4 présente un état de l'art des outils de vérification de programmes en particulier l'analyse statique et la dépendance causale ;
- Le chapitre 5 détaille notre première contribution durant cette thèse qui consiste en une API de filtrage de données compatible avec la spécification Java Card 3 et un outil de détection de vulnérabilités XSS basé sur une analyse statique du bytecode.
- Le chapitre 6 présente l'implémentation de notre outil d'analyse et des résultats expérimentaux.

La troisième et dernière partie aborde notre deuxième axe de recherche qui concerne la sécurité des cartes à puce à serveur Web embarqué au niveau du protocole HTTP implémenté. Elle est composée de trois chapitres :

- Le chapitre 7 décrit le protocole HTTP et les différentes techniques de vérification et de validation de protocoles. Nous nous attardons particulièrement sur la techniques de *fuzzing* que nous avons adoptée ;
- Le chapitre 8 détaille notre seconde contribution durant cette thèse qui consiste en un outil de *fuzzing* qui comporte un ensemble d'optimisations rendant le *fuzzing* plus intelligent ;
- Le chapitre 9 décrit notre implémentation de l'outil de *fuzzing* et les résultats expérimentaux obtenus.

Nous finissons ce rapport par une conclusion générale et de quelques perspectives pour des travaux futurs.

Première partie

Domaine d'étude

Chapitre 1

Contexte d'étude

Sommaire

1.1 La carte à puce **9**
1.2 Historique **10**
1.3 Les protocoles de communication **11**
1.3.1 Le protocole APDU 12
1.3.2 Le protocole BIP 12
1.3.3 Le protocole HTTP 13
1.4 Les cartes à puce à serveur Web embarqué **13**
1.4.1 Les cartes à serveur Web SCWS 14
1.4.2 Les cartes Java Card 3 Edition Connectée 17
1.5 Cas d'utilisation des cartes à puce à serveur Web embarqué **21**
1.5.1 Accès rapide et simple aux services offerts par la carte 21
1.5.2 Amélioration de l'interface d'accès aux services offerts par l'émetteur . . 22
1.5.3 Sécurité de connexion d'un terminal distant à une application . . . 22
1.5.4 Exemple d'application : « le Disque Virtuel » 23
1.6 Conclusion **24**

Dans ce chapitre nous présentons notre domaine d'étude, à savoir, la carte à puce à serveur Web embarqué. Avant de passer à la description de ce type de cartes, nous commençons par une brève définition de la carte à puce et un historique de ses évolutions les plus marquantes puis nous présentons l'ensemble des différents protocoles utilisés dans les cartes à serveurs Web embarqué. Nous finirons par présenter l'apport de l'intégration d'un serveur Web dans une carte à puce à travers quelques cas d'utilisation et un exemple d'application Web dédiée à ce type de cartes.

1.1 La carte à puce

Une carte à puce est un composant électronique, défini par un ensemble de normes établies par l'organisme international de normalisation (ISO), portant le nom de ISO 7816 [Sta]. Elle permet d'effectuer des opérations (stocker, calculer, etc.), de façon sécurisée. Les cartes à puce sont

utilisées dans différents domaines (bancaire, téléphonie, etc.). Sa puissance de calcul et sa capacité de stockage d'informations sont restreintes et peuvent être différentes d'une carte à l'autre.

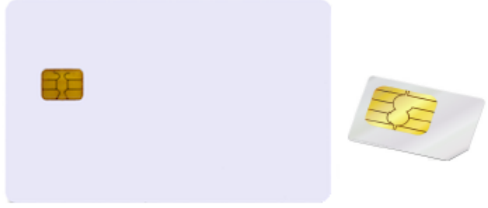

Fig. 1.1 – Carte à puce

Il existe différentes plateformes logicielles permettant d'utiliser les cartes à puce. Java Card est l'une des plateformes les plus utilisées sur le marché, elle représente à l'heure actuelle 95% des cartes à puce dans le monde. Java Card est un sous-ensemble des technologies Java adapté aux objets à faibles ressources comme les cartes à puce. Il s'agit d'une plateforme portable, sécurisée et multi-applicative qui intègre les avantages du langage Java :

- la programmation orientée objet offerte par Java (contre l'assembleur auparavant) ;
- une plateforme ouverte qui définit des interfaces de programmation (APIs [1]) et un environnement d'exécution standardisé ;
- l'utilisation d'une machine virtuelle permettant la portabilité des applications et une sécurité accrue lors de l'exécution du code ;
- une plateforme qui encapsule la complexité sous-jacente (assembleur) et les détails du système des cartes à puce.

Grâce à la machine virtuelle Java Card (JCVM), il est possible de charger dans la carte plusieurs applications (Applets) capables de s'exécuter dans un environnement sécurisé. Les programmes chargés dans cette plateforme sont généralement des applets bien que depuis la version 3.0 édition connectée il est possible de charger des servlets aussi. La compilation des applets Java génère des fichiers CAP [2] (plateforme Java Card classique) ou des fichiers class (Java card 3.0 édition connectée).

1.2 Historique

Depuis leur invention, les cartes à puce ont connu une très grande évolution, commençant par de simples cartes à mémoire, suivies par des cartes à microprocesseur, puis des cartes multi-applicatives et enfin la plus récente révolution est la naissance des cartes à puce à serveur Web embarqué. Dans ce qui suit, nous présentons un bref historique des évolutions les plus marquantes :

- **1979 :** Sortie des premières cartes à microprocesseur : La Bull CP8 (1 Ko de mémoire programmable et un cœur à base de microprocesseur 6805) ;
- **1983 :** Utilisation des cartes à puce dans le secteur médical et social. Dans la même année, apparaissent les premières cartes téléphoniques à mémoire ;

1. *Application Programming Interface* ou API est un ensemble de fonctions, procédures ou classes mises à disposition par une bibliothèque logicielle

2. Format pré-chargé des applications Java Card (applets), généré en dehors de la carte par un outil appelé *convertisseur* et à partir de fichier class. Ce format est utilisé pour des raisons d'optimisation des ressources limitées des cartes standards

- **1984 :** Commercialisation des premières cartes à mémoire bancaires, conçues par Bull, Schlumberger et Philips.
- **1985 :** Bull livre ses premières cartes bancaires dotées de microprocesseur ;
- **1991 :** Lancement des premiers réseaux et services GSM (roaming, SMS, etc.), et des premières cartes *Subscriber Identity Module* (SIM) ;
- **1996 :** Création du langage Java Card, un sous ensemble du langage Java ;
- **1997 :** Bull, Gemplus et Schlumburger créent le Java Card Forum afin de discuter et de proposer de nouvelles spécifications. Java Card devient la plateforme standard de développement des applications pour cartes à puce. La Java Card a connu plusieurs versions, nous citons particulièrement :
  - la Java Card 2.1, sortie en 1999, qui inclut des modifications de quelques APIs, notamment au niveau de la cryptographie et des exceptions.
  - la version Java Card 2.2 (en 2002) a pour principales nouveautés, la prise en charge des canaux logiques et l'appel de méthodes à distance.
  - la Java Card 2.2.2 est une mise à jour de la précédente ; elle inclut une amélioration de l'interopérabilité avec de multiples interfaces de communication et des dispositifs de sécurisation normalisés. Elle fournit également de nouvelles APIs pour un développement plus économique des applications. Cette version a été adaptée pour correspondre aux normes des cartes à puce USIM [3], tout en étant complètement compatible avec les versions précédentes.
- **2002 :** la technologie de communication *near field communication* NFC a eté lancée par Sony et Philips. C'est une technologie de communication sans-fil à courte portée et haute fréquence, permettant l'échange de données entre un lecteur et un terminal mobile ou entre les terminaux eux-mêmes à un débit maximum de 424 Kbits/s et à une distance de 20 cm au maximum.
- **2008 :** l'organisation *Open Mobile Aliance* (OMA) [All08a, All08c, All08b], a annoncé la spécification d'un serveur Web *Smart Card Web server* (SCWS), destiné à des cartes SIM permettant d'exécuter des applications plus riches sur les cartes à puce, et communiquant sur le réseau via le protocole *Bearer Independent Protocol*(BIP).

  Dans la même année en mars 2008, Sun (désormais Oracle) annonce la spécification Java Card 3.0, dont la plus récente version est Java Card 3.0.2 en 2010. Elle est définie en deux éditions distinctes, *l'édition classique* et *l'édition connectée*.
  - *L'édition classique* est semblable à la version Java Card 2.2, avec quelques améliorations aux niveaux des APIs notamment au niveau cryptographique.
  - *L'édition connectée* définit un nouveau modèle de programmation, les servlets et le support d'API et de fonctionnalités additionnelles telles que, le multi-taches (*Multi-threading*), les types String, le ramasse-miette, etc. Nous présentons cette version plus en détail dans la section 1.4.2.

1.3 Les protocoles de communication

La communication entre une carte à puce classique et un terminal se fait via un protocole de communication appelé *Application Protocol Data Unit* (APDU). L'intégration d'un serveur Web embarqué dans les cartes à puce nécessite le support du protocole HTTP permettant l'échange

3. Carte à puce dédiée à la téléphonie 3G

de données entre les applications et le navigateur Web. D'autre part, les cartes à puce à serveur Web embarqué sont considérées comme une entité d'un réseau. TCP/IP sont les protocoles de communication réseau les plus utilisés ; cependant, ils ne peuvent être supportés que sur des cartes à hautes ressources. Le protocole BIP est le plus adapté pour des cartes moins performantes. Dans ce qui suit nous présentons brièvement cet ensemble de protocoles.

1.3.1 Le protocole APDU

Le protocole APDU est le protocole de communication défini dans la norme ISO 7816-4, permettant à une applet s'exécutant sur une carte à puce de recevoir des requêtes émanant du terminal et d'envoyer des informations en réponse. Le dialogue entre la carte et le terminal se fait via un échange de commandes et réponses APDU.

Une commande APDU est une requête envoyée par le terminal vers une applet chargée dans une carte à puce. La *figure 1.2* représente les différents champs qui la composent :

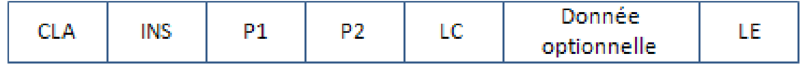

Fig. 1.2 – Commande APDU

- **CLA** : octet de classe défini par l'ISO 7816 comme fournissant un certain nombre d'informations sur la commande (exemples : 0xBC pour les cartes vitales, 0x0A pour les cartes SIM, 0x00 pour les Mastercard/Visa) ;
- **INS** : octet d'instruction ;
- **P1** : paramètre 1 ;
- **P2** : paramètre 2 ;
- **Lc** : taille des données envoyées à la carte (0x00 si pas de données) ;
- **Le** : taille des données attendues en réponse.

La réponse APDU contient des informations sur le résultat de la commande et éventuellement des données si nécessaires. Elle est composée d'un champ de donnée et d'un mot d'état (*figure : 1.3*). La donnée doit respecter la taille indiquée dans le champ Le de la commande. Le mot d'état (*status word*) est codé sur deux octets : SW1 et SW2. SW1 indique si la commande à été correctement exécutée et SW2 correspond à des informations supplémentaires concernant la réponse. A titre d'exemple, la valeur du mot d'état : 90 00 signifie que la commande s'est exécutée correctement.

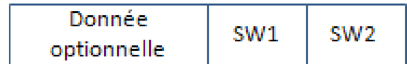

Fig. 1.3 – Réponse APDU

1.3.2 Le protocole BIP

Le protocole de communication BIP a été introduit dans la spécification ETSI [ETS10a], il est dédié aux cartes de type (U)SIM. Il permet à une carte (U)SIM d'utiliser les moyens de communication de l'équipement (téléphone) auquel elle est connectée, quel que soit la technologie de

communication utilisée. Il offre à la fois la possibilité d'accès à des données portées par des interfaces (Bluetooth, IrDA, etc.) et à des données réseau (GPRS, 3G, etc). BIP est défini par un ensemble de commandes proactives permettant d'ouvrir les canaux de communication, d'émission et de réception des données.

- `OpenChannel` : ouvre un canal de communication entre la carte et le téléphone ;
- `GetChannelStatus` : demande au téléphone l'état du canal de communication ;
- `SendData` : envoie des données au téléphone ;
- `ReceiveData` : se prépare à recevoir des données ;
- `CloseChannel` : ferme un canal de communication.

La communication avec le terminal se fait via des protocoles standards UDP ou TCP (*figure : 1.4*). BIP est le protocole de communication utilisé dans les cartes à puce *Smart Card Web Server*, en remplacement du protocole TCP/IP qui n'est pas supporté. Nous présentons ce type de cartes dans la section suivante.

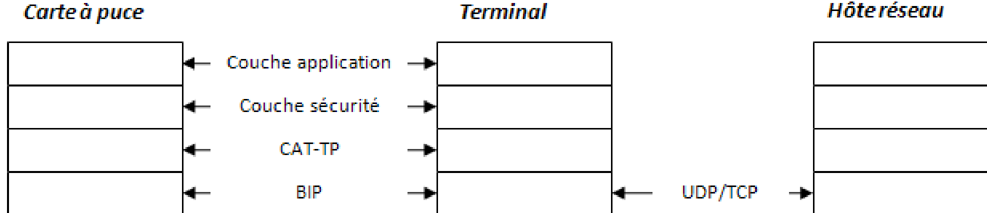

Fig. 1.4 – Communication entre SCWS et téléphone mobile via le protocole BIP

1.3.3 Le protocole HTTP

Hypertext Transfer Protocol (HTTP) est le protocole de communication le plus utilisé sur Internet. Il est basé sur le modèle de communication client/serveur, le client HTTP correspondant au navigateur Web à partir duquel un utilisateur peut envoyer des requêtes à un serveur contenant des données. La version HTTP/1.1 est décrite dans la RFC 2616 datée de juin 1999. Les cartes à puce à serveur Web embarqué utilisent ce même protocole permettant aux applications Web de la carte d'échanger des requêtes/réponses avec le client Web hébergé dans le terminal (le téléphone mobile en l'occurrence). Dans la partie III de ce document, nous présentons plus en détail ce protocole et nous proposons un outil de *fuzzing* pour tester et vérifier la conformité et la robustesse du protocole HTTP implémenté dans une carte à puce.

1.4 Les cartes à puce à serveur Web embarqué

La stratégie des fabricants de cartes à puce a toujours été d'inventer des objets de plus en plus puissants et de nouveaux standards pour chaque nouvelle fonction. Avec l'arrivée du protocole BIP, il est devenu possible d'établir une communication entre la carte SIM et le terminal en utilisant le protocole *Internet Protocol* (IP). Ainsi, l'organisation OMA a réalisé la spécification de SCWS, un serveur Web HTTP embarqué dans des cartes SIM. Gemalto a ensuite fabriqué les premiers produits de ce type de cartes SIM, capables de servir de serveur Web (SCWS), de sauvegarder l'ensemble

du contenu du téléphone, d'implémenter des fonctions NFC voir GPS indépendantes de celles du téléphone etc. D'autre part, Sun (aujourd'hui Oracle) a voulu exploiter les nouvelles évolutions matérielles des cartes à puce, en annonçant les spécifications Java Card 3 *édition connectée*, une nouvelle version de la plateforme Java Card. Cette plateforme est destinée à des cartes à puce de haut de gamme. Elle supporte de nombreuses fonctionnalités et APIs la rendant très proche de Java. La communication sur le réseau se fait via les protocoles TCP/IP qui sont cette fois directement supportés sur cette plateforme.

1.4.1 Les cartes à serveur Web SCWS

Le serveur Web SCWS est la technologie normalisée par l'organisation OMA et dédiée aux cartes SIM [All08d, All08a, All08c, All08b]. La spécification SCWS/1.0 définit un serveur Web HTTP/1.1 [All08a] intégré dans la carte à puce dont le contenu est accessible via une URL. Un protocole d'administration est également défini permettant de mettre à jour et de configurer à distance le SCWS, ce protocole est aussi basé sur le protocole HTTP/1.1.

Dans la version SCWS/1.1, la gestion à distance par différentes entités de confiance est optimisée. Chaque entité peut être contrôlée pour vérifier à quel contenu et quelle application elle a le droit d'accès en utilisant une URL, à partir d'un client HTTP/HTTPS installé sur son terminal distant.

Le premier produit de cartes SCWS a été réalisé par Gemalto qui a développé des cartes SIM à SCWS combiné à la technologie *Near Field Communication* (NFC), sur la base d'une plateforme Java Card 2.2 [Gem]. Ils ont également proposé le protocole *Single Wire Protocol* (SWP) défini dans le standard *European Telecommunications Standards Institute* (ETSI), TS 102 588. Ce protocole permet à la carte de communiquer sur un seul fil avec un circuit NFC dans n'importe quel "objet intelligent". SWP n'utilise qu'un seul contact, il permet ainsi de libérer les deux contacts restants qui pourraient être utilisés pour établir une connexion de type USB avec le téléphone pour communiquer en mode IP. En novembre 2008, Gemalto et LG Electronics ont annoncé le lancement du premier téléphone mobile commercial avec un serveur Web embarqué sur une carte à microprocesseur. Le SCWS n'est pas encore une technologie très répandue, mais elle pourrait révéler tout son potentiel avec déploiements de la technologie NFC.

Architecture du SCWS

Le SCWS comprend trois composantes :
- un serveur HTTP,
- un dépôt (*repository*) contenant les données SCWS,
- un gestionnaire de tâche d'administration qui met à jour le dépôt SCWS.

L'entité de communication externe (navigateur Web ou application OTA) communique avec la carte en utilisant les requêtes/répenses HTTP. Après être relayée à la couche de communication, la requête HTTP est envoyée au serveur HTTP. La gestion des multi-requêtes est assurée par la couche communication, le serveur étant censé recevoir et gérer une requête par session. Une session SCWS commence avec la réception de la requête et termine par l'émission de la réponse. Le serveur HTTP se charge de transférer la requête à la destination correspondant à l'URL indiquée dans l'entête de la requête. Cette URL peut pointer vers une ressource dynamique (servlet dans le cas de la plateforme Java Card 2.2), une ressource statique, ou une tache administrative (*figure : 1.5*).

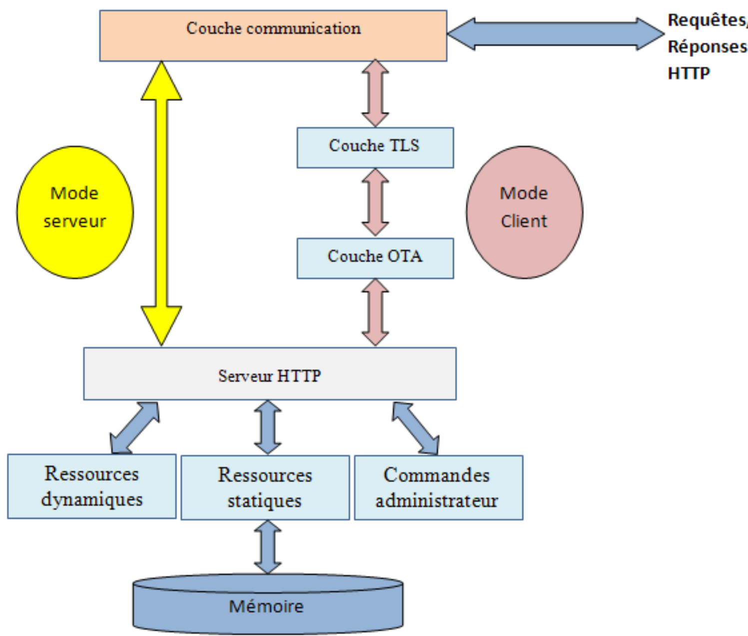

Fig. 1.5 – Architecture du SCWS

Les spécifications de l'OMA définissent un contrôleur d'accès HTTP qui s'appuie sur des ensembles protégés. Un ensemble protégé contient des ressources et la liste des utilisateurs qui sont autorisés à y accéder. Si le chemin d'accès indiqué dans l'URL de la requête correspond à un ensemble protégé alors l'accès à la ressource demandée est autorisée uniquement aux utilisateurs définis dans par ensemble.

Des politiques de sécurité ont été mises en place pour prévenir un accès illégal aux ressources de la carte [NN11] :

1. séparation des URLs : les ressources statiques et les différentes servlets doivent être séparées en terme d'URL correspondant à chacune d'elles.
2. pas d'accès illégal à une ressource statique : l'accès ne doit pas être possible à :
   - une ressource qui n'a pas de correspondance URL ;
   - une ressource qui ne correspond pas à l'URL indiquée dans la requête ;
   - une ressource non autorisée.
3. pas d'invocation illégale de servlets : une invocation ne peut pas cibler :
   - une servlet non enregistrée ;
   - une servlet dont aucune correspondance n'est définie ;
   - les servlets qui ne correspondent pas à l'URL indiquée dans la requête ;
   - des servlets non autorisées.
4. pas d'accès aux ressources externes au serveur : les données de la carte à puce qui sont externes au serveur Web ne doivent pas être accessibles.
5. ne pas envoyer des ressources en cas d'erreur : en cas d'erreur, la réponse ne doit pas contenir autre que le code d'erreur.
6. gestion sécurisée du contenu de la carte : les contenus de la carte (ressources statiques, applets, dépôt) sont mis à jour uniquement par l'administrateur.

Les cartes à serveur Web basées sur la plateforme Java Card

L'architecture d'une carte à puce basée sur la plateforme Java Card 2.2 et contenant un serveur SCWS est présentée dans la *figure 1.6*. Dans le but d'avoir des applications interopérables fournissant du contenu dynamique, les spécifications ETSI TS.102.588 [ETS10b] définissent une API et des bibliothèques disponibles pour la plateforme Java Card 2.2. Ces bibliothèques permettent de développer des applications Web basées sur des servlets et fournissent des méthodes permettant à une servlet de traiter les requêtes HTTP et de retourner des réponses au client (navigateur Web)(exemple doPost() pour traiter les requêtes POST). Les servlets sont des Applets enregistrées dans le SCWS.

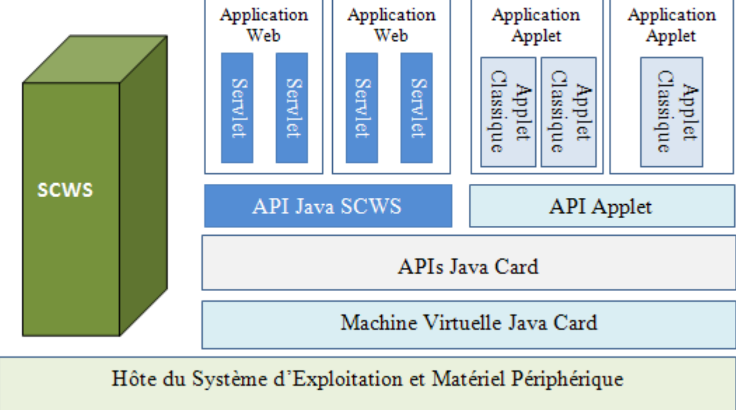

Fig. 1.6 – SCWS dans une carte Java Card 2.2

Communication entre le SCWS et le terminal

Le SCWS est utilisé en mode serveur lorsqu'il communique avec le client (navigateur Web) du téléphone et en mode client lorsqu'il est administré à distance par l'émetteur de la carte [All08c, GDS07]. Les protocoles TCP/IP n'étant pas supportés dans la plupart des cartes à puce, une passerelle SCWS doit être installée dans le terminal pour assurer la communication entre le terminal et le protocole de transport local installé sur la carte à puce. La passerelle SCWS se charge de traduire les requêtes HTTP reçues par le client HTTP en commandes spécifique au protocole local (BIP en l'occurrence). Elle assure aussi un contrôle d'accès sécurisé aux données du SCWS.

Le protocole BIP est le protocole de transport local le plus utilisé. La carte communique avec le téléphone à l'aide de commandes BIP [GDB06]. Ces commandes sont constituées de un ou plusieurs TLV (*Tag Length Value*) correspondant à un tag, la taille des données et les données. Elles font parties de la technologie *SIM Application Toolkit* (SAT) définie par l'ETSI. Le protocole BIP seul ne suffit pas pour assurer la transmission des paquets. Deux protocoles de transport sont également utilisés :

- `TCP` est déployé sur le téléphone et utilisé lorsque la carte SIM communique avec le téléphone ;
- `CAT-TP` (*Card Application Toolkit Transport Protocol*) est déployé sur la carte SIM et utilisé lorsque la carte communique avec un serveur distant.

Le terminal utilise une passerelle BIP permettant d'intercepter les commandes BIP envoyées par la carte et de les transformer en commandes TCP afin de les renvoyer à l'application HTTP.

Cette passerelle joue le rôle d'un convertisseur de protocole. Pour communiquer avec la passerelle, la carte dispose des commandes *BIP SIM Toolkit*. À chaque commande *Sim Toolkit*, le téléphone répond via une commande de type `TerminalResponse`. De plus, la passerelle interagit avec la carte à l'aide d'évènements :

- `DataAvailable` : commande d'initiation d'un envoi de données du terminal à la carte.
- `ChannelStatus` : commande pour demander à la carte, l'état du canal de communication.

Toute communication entre la carte et le téléphone débute par l'ouverture d'un canal de communication. Pour cela, le téléphone envoie la commande `TerminalProfile` afin d'informer la carte qu'il utilise le protocole BIP. Après cette commande, la carte attend la commande *Fetch* avant d'ouvrir un canal de communication avec le téléphone via la commande `OpenChannel`. La commande `Fetch` initie une communication avec le terminal. Si le terminal répond correctement via la commande `TerminalResponse`, le canal de communication est ouvert. Ensuite, la passerelle BIP écoute sur le port 3516 dédié aux communications HTTP et l'utilisateur peut interagir avec le SCWS à partir de son navigateur Web en utilisant l'adresse http ://127.0.0.1 :3516/index.html.

1.4.2 Les cartes Java Card 3 Edition Connectée

La Java Card 3 est la dernière version de la plateforme Java Card, dont la plus grande évolution est *l'édition connectée*. Elle apporte le support d'un nouveau modèle de programmation basé sur des servlets et l'ouverture de la plateforme au réseau IP, via un serveur Web intégré et le support des protocoles standards TCP/IP et HTTP. Cependant, ces nouvelles fonctionnalités font que cette spécification n'est supportée que sur des cartes à puce haute de gamme (un processeur 32 bits, 128 ko d'EEPROM, 512 ko de mémoire ROM, 24 ko de RAM). Dans la suite, nous désignons par le nom Java Card 3, l'édition connectée.

Architecture de la plateforme

La plateforme Java Card 3 introduit dans son architecture une nouvelle machine virtuelle et un nouvel environnement d'exécution qui supporte, en plus des applets classiques de la version Java Card 2.2, des applets étendues et les applications Web basées sur des servlets. Ces différentes applications se caractérisent respectivement par :

- *Les Applets classiques* :
  - communiquent avec le terminal via le protocole APDU (norme ISO 7816-4).
  - peuvent implémenter uniquement les API compatibles avec la précédente version.
- *Les Applets étendues* :
  - communiquent avec le terminal via le protocole APDU.
  - supportent les API des applets classiques et aussi de nouvelles APIs telles que les Threads, Strings, et GCF (*Generic Connection Framework*).
- *Les Servlets* :
  - communiquent avec le terminal via le protocole HTTP ou HTTPS.
  - Basées sur la spécification des Servlets version 2.4 [Ser03].

La plateforme Java Card est conçue pour des cartes à puce possédant des interfaces physiques à haut débit. Les protocoles vers des interfaces d'E/S additionnelles ont été intégrés, comme :

- Universal Serial Bus (USB), MultiMediaCard (MMC)

- ISO 14443 (contactless)
- Le protocole de communication par APDU basé sur la norme ISO 7816-4 est aussi supporté.

Au niveau logique, la mise en œuvre de la plateforme Java Card 3 doit fournir aux applications une interface réseau logique qui prend en charge les protocoles réseau suivants :

- *Internet Protocol* (IP)
- *Transmission Control Protocol* (TCP)
- *Universal Datagram Protocol* (UDP)
- *Transport Layer Security* (TLS)
- *HyperText Transfer Protocol* (HTTP)
- *Secure HyperText Transfer Protocol* (HTTP sur TLS)

La bibliothèque Java Card 3

En plus du nouvel environnement d'exécution, la plateforme Java Card se caractérise par une bibliothèque beaucoup plus riche qui réduit nettement l'écart entre les langages Java Card et Java. Parmi les éléments Java qui sont désormais supportés nous pouvons citer :

- tous les types Java de base exceptés les nombres à virgule flottante (float et double) ;
- les structures de données de base, à savoir les tableaux multidimensionnel et les objets de type String et StringBuffer
- les supports natifs des fichiers classes avec l'édition des liens dans la carte ;
- les classes utilitaires : un sous ensemble important du paquet java.util (Stack, Vector, Hashtable, Enumeration, Iterator, Date, Calendar, TimeZone, EventObject, EventListener, ResourceBundle, ListResourceBundle, Random, StringTokenizer, etc) ;
- la bibliothèque GCF (*Generic Connection Framework*) permettant d'ouvrir des connexions réseau et le paquetage microedition.io, composé de classes qui gèrent des connections génériques. Le paquetage java.io est en partie supporté incluant un ensemble de classes nécessaires à la création, la lecture, l'écriture et le traitement des flux ;
- la persistance dans Java Card 3 est différente des précédentes versions. Un objet est dit persistant s'il est défini dans un champ de type `Static`, instance de `javacard.framwork.applet` ou instance de `javax.Servlet.ServletContext` ou par accessibilité (si un champ persistant est affecté à un champ volatile, ce dernier devient automatiquement persistant) ;
- les transactions ont été améliorées, les transactions Java basées sur les annotations sont supportées ;
- le Multi-tâches, permettant à un ensemble d'applications de s'exécuter simultanément. Cependant les groupes de threads ou les threads fonctionnant comme des démons, ne sont pas supportés.
- la destruction automatique des objets non utilisés ;
- etc.

Les applications Web Java Card

Une application Web dans Java Card est une archive possédant l'extension `.war` qui est ensuite déployée et instanciée dans le conteneur Web. Elle est identifiée par le chemin spécifique (root path) par lequel elle est adressée dans ce conteneur Web. Ce chemin est utilisé :

– comme identificateur unique (URI application) de l'application sur la partie interne de la plateforme pour la communication inter-application ;
– comme identificateur externe (URL) pour l'envoi de requêtes HTTP par un client Web (navigateur Web).

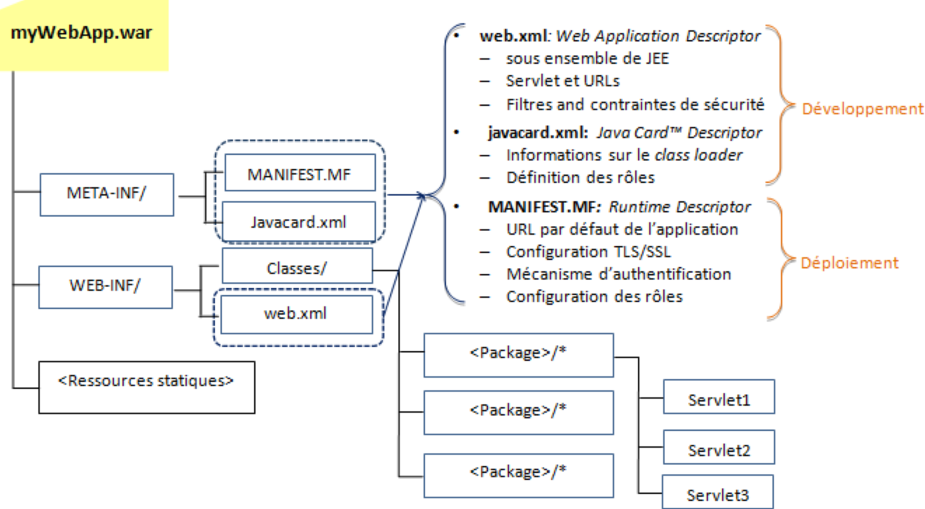

Fig. 1.7 – Composents d'une application Web Java Card

Structure de fichiers contenus dans une archive .war Une application Web Java Card a la même structure (répertoires, fichiers de configuration, etc) que les applications Web JEE (*figure : 1.7*. La seule restriction est qu'il n'existe pas de bibliothèque spécifique à chaque application Web (le répertoire WEB-INF/lib n'existe pas). De plus, la plateforme Java Card 3 définit un descripteur d'application spécifique nommé javacard.xml [JC309c]. Le répertoire WEB-INF contient les classes de l'application Web et un descripteur de déploiement (web.xml). Les fichiers `.class` et les servlets sont stockés dans des paquetages. Ils sont accessibles via un chemin d'accès relatif au nom du paquetage (*package*) qui les héberge.

Une servlet est une classe Java permettant la génération dynamique de contenu. Elle reçoit des requêtes HTTP (sous forme d'objets Java). La spécification des servlets pour la plateforme Java Card est un sous ensemble de la spécification des servlets Java v 2.4. Des fonctionnalités non supportées par la JCVM ont été supprimées telles que les pages JSP (Java Server Pages).

Le fichier META-INF/MANIFEST.MF définit des informations supplémentaires du descripteur. Le répertoire META-INF contient également le descripteur d'application javacard.xml spécifique à la plateforme Java Card.

Tous les autres fichiers inclus dans le fichier .war, en dehors de la hiérarchie des répertoires META-INF et WEB-INF, ne sont pas directement utilisés par la plateforme. Ils sont simplement mémorisés sous forme de fichiers ressources de l'application et correspondent à des ressources Web statiques desservies par le Web conteneur pour un client off-card.

La sécurité de la plateforme Java Card 3

Comme ses précédentes versions, la Java Card 3 *édition connectée* permet de bénéficier des avantages du langage Java, principalement :

- le typage : Java est un langage fortement typé permettant d'éviter les codes frauduleux ;
- les pointeurs ou références : les opérations arithmétiques sont interdites sur les pointeurs ;
- la conversion de type : les conversions de type sont encadrées avec des concepts de polymorphisme, d'encapsulation et d'héritage.

Outre ces avantages, la plateforme offre des mécanismes de sécurité dont certains étaient déjà présents dans les précédentes versions et d'autres ont été renforcés. Parmi ces différents mécanismes, nous citons :

- **La récupération de mémoire** : un mécanisme **automatique** appelé ramasse-miettes (*Garbage Collector*) permet de libérer les zones mémoires des objets volatiles (objets à durée de vie limitée stockés dans une mémoire volatile) non référencés depuis une longue durée. Pour des objets persistants (stockés dans une mémoire non-volatile) ce mécanisme peut être appelé à la demande.
- **L'isolation des objets des applications grâce à un pare-feu (firewall)** : le pare-feu est un mécanisme permettant de partitionner le système en espaces protégés où chaque partie appartient à un contexte de groupe différent. Un contexte de groupe est un paquetage Java composé d'un ensemble d'applets instances de classes de même paquetage. Un objet ne peut être accessible que par des méthodes de même contexte de groupe. Toutefois, la plateforme offre un mécanisme SIO (*Shareable Interface Object*) permettant à une application de disposer d'une interface de partage, pour rendre accessibles certains services à des applications autorisées de contextes de groupe différents. Il existe aussi un contexte propre à la plateforme appelé le contexte JCRE (*Java Card Runtime Environnement*). Une application ne peut accéder qu'aux services (APIs) de la plateforme pour lesquelles elle a les permissions requises. Cependant les applets appartenant au contexte JCRE peuvent accéder à tout objet de n'importe quel autre contexte de la plateforme.
- **La vérification obligatoire de bytecode** : le vérificateur de bytecode était optionnel dans les précédentes versions. Il est maintenant obligatoire lors du chargement des applets sur la plateforme. Il consiste à faire une interprétation abstraite du bytecode qui implique entre autre de verifier le type des objets manipulés, et la consistance de la pile.
- **La gestion de données volatiles** : certains objets peuvent être conservés en RAM, ce qui peut rendre certaines attaques plus complexes.
- **Un contrôle d'accès fin (*Access Controller*)** : permet de protéger l'accès aux services de la plateforme et aux ressources d'une application par d'autres applications. Lors du chargement d'une application un fichier *policy* est joint : il comporte toutes les permissions associées à l'application. Quand une application tente d'accéder à un service, le système récupère le contexte d'exécution de l'application et les permissions qui lui sont associées pour vérifier si oui ou non cette application est autorisée à accéder à ce service.
- **Une communication sécurisée basée sur TLS** : la communication sur le réseau est sujette à de nombreuses attaques telles que l'écoute de la transmission ou l'injection de paquets, etc. TLS (*Transport Layer Security*) [HTT] permet de sécuriser cette communication en créant un tunnel sécurisé entre le client et le serveur basé sur des algorithmes cryptographiques.
- **Une authentification des utilisateurs pour une gestion fine de l'accès aux res-**

sources : en effet la plateforme offre un mécanisme permettant de vérifier l'identité de l'utilisateur et ses droits d'accès : les rôles. Un rôle détermine les droits d'accès d'un utilisateur authentifié à des ressources protégées, telles que les services basés sur les SIO et les ressources Web.
Dans la Java Card, deux types d'utilisateurs sont définis : le propriétaire de la carte et d'autres utilisateurs (comme l'administrateur). Chaque utilisateur a une identité et des droits différents sur la carte.
L'authentification se fait via des identificateurs appelés *authenticators*. Un identificateur est le service spécialisé dans l'authentification qui peut utiliser différents schémas d'authentification tels qu'un mot de passe, un code PIN ou une information biométrique. Ce service peut être utilisé par une application ou un conteneur Web.

– **Les annotations de sécurité** : elles sont supportées dans Java Card 3 ; il s'agit d'un mécanisme permettant de sécuriser tout ou une partie de l'application en définissant une politique d'exécution. Les annotations peuvent être utilisées pour protéger les applications contre des comportements inattendus de la plateforme. Elles peuvent être associées à des classes ou des méthodes. La plateforme fournit un ensemble d'annotations telle que `@Sensitive.Confidentiel` qui signifie que la méthode ou la classe ne doit être exécutée que dans un contexte authentifié. Cependant, les fournisseurs de la carte ou les développeurs du système ont la possibilité d'implémenter des annotations additionnelles.

1.5 Cas d'utilisation des cartes à puce à serveur Web embarqué

L'intégration d'un serveur Web embarqué introduit de nouvelles problématiques au domaine des cartes à puce. Vis-à-vis des évolutions des téléphones mobiles qui permettent d'accéder à Internet et à tous les services offerts sur ce réseau, nous pouvons nous demander l'utilité même d'embarquer un serveur Web dans la carte à puce. Dans ce paragraphe. Nous répondons à cette interrogation par un ensemble de cas d'utilisation [KICLB10, KMM10].

1.5.1 Accès rapide et simple aux services offerts par la carte

L'utilisation des protocoles standards d'Internet offrent une plus grande flexibilité d'utilisation et d'intégration de la carte à puce aux équipements existants et aussi une plus grande aisance d'administration pour les émetteurs de carte (industriels et opérateurs). Il est désormais possible de lire le contenu de sa carte, ou de faire des transactions bancaires ou commerciales en toute sécurité à partir de n'importe quel terminal contenant un navigateur Web. Les applications sont hébergées dans un environnement sécurisé (la carte à puce) et sont donc mieux protégées que sur un terminal qui peut facilement être compromis par un virus ou un programme malicieux. L'émetteur de la carte peut installer plusieurs services accessibles localement sur la carte via un navigateur Web comme tout autre service d'Internet. L'accès à Internet ou à un service disponible sur la carte est donc transparent pour l'utilisateur. Il peut télécharger les applications qu'il souhaite en accédant à une page d'accueil qui lui propose une liste de services et des contenus sur Internet. Ces applications peuvent également être configurées selon ses besoins. En demandant d'installer une

application, le propriétaire de la carte remplit un formulaire et l'application installée sera configurée selon ses besoins indiqués dans le formulaire.

Du côté de l'émetteur de la carte, l'utilisation d'un serveur Web embarqué et du protocole HTTP(S) standard permet une administration à distance des applications installées sur la carte. En effet, un administrateur peut effectuer un suivi des applications, faire des mises à jour ou des installations, à condition que la carte soit connectée à l'Internet.

Le serveur Web embarqué est une solution à l'amélioration de la portabilité des applications et des services en cas de renouvellement de l'équipement mobile. En effet, en cas de changement de téléphone, il ne serait plus nécessaire de recharger les applications utilisateur, tandis qu'il faudrait probablement adapter le contenu à la taille de l'écran du nouvel appareil.

1.5.2 Amélioration de l'interface d'accès aux services offerts par l'émetteur

L'utilisateur souhaite avoir une interface d'accès aux services offerts par la carte et qui soit conviviale et facilement personnalisable. Les standards du Web permettent d'avoir une expérience utilisateur continue et équivalente à celle disponible sur Internet. En utilisant des cartes standards, ces interfaces sont offertes par les lecteurs de carte tels que les téléphones mobiles et elles diffèrent d'un appareil à l'autre. L'introduction d'un nouveau modèle d'application complètement dédié au mode Web (servlet) offre aux opérateurs ou émetteurs de cartes la possibilité de concevoir des interfaces conviviales et standardisées pour tous les lecteurs à condition que celui-ci dispose d'un navigateur Internet. Dans le cas d'une carte SIM, par exemple, une gestion locale (dans la carte à puce) du carnet d'adresses via une interface est facilement réalisable. De plus la forte connectivité de la carte permet d'envisager un enrichissement de ce genre d'applications.

1.5.3 Sécurité de connexion d'un terminal distant à une application

Une carte à puce à serveur Web embarqué est une carte à forte connectivité avec une possibilité d'accéder à Internet via un réseau intermédiaire (par exemple, les réseaux 2G/3G pour les cartes SIM). Elle permet également de renforcer l'accès à distance à des applications critiques. En effet, sa propriété de *tamper-resistance* permet de garantir l'identité de l'utilisateur qui tente d'accéder à distance à des données critiques (exemple : application bancaire).

D'autre part, l'aspect Web apporte l'utilisation du protocole TLS pour la sécurisation de bout-en-bout entre la carte et son correspondant, ainsi qu'une gestion fine des utilisateurs (définition des rôles et des permissions), de leur authentification et de l'accès aux différentes ressources de la carte. S'ajoute à cela d'autres aspects sécuritaires qui ont été renforcés et rajoutés dans la Java Card 3 (vérificateur de bytecode, gestion de données volatiles, récupération de mémoire).

La carte à puce à serveur Web permet également de résoudre le problème de déploiement d'applications dans un environnement non contrôlé comme un téléphone portable, où des problèmes tels que la suppression accidentelle d'une application critique par un utilisateur final peut se produire.

Toutefois, des failles dans le développement des applications Web peuvent ouvrir des portes à différents types d'attaques Web, tel que l'injection de données malicieuses dans une page Web, le vol d'identifiants de session ou l'accès non autorisé à des ressources [OWA07]. Un attaquant peut

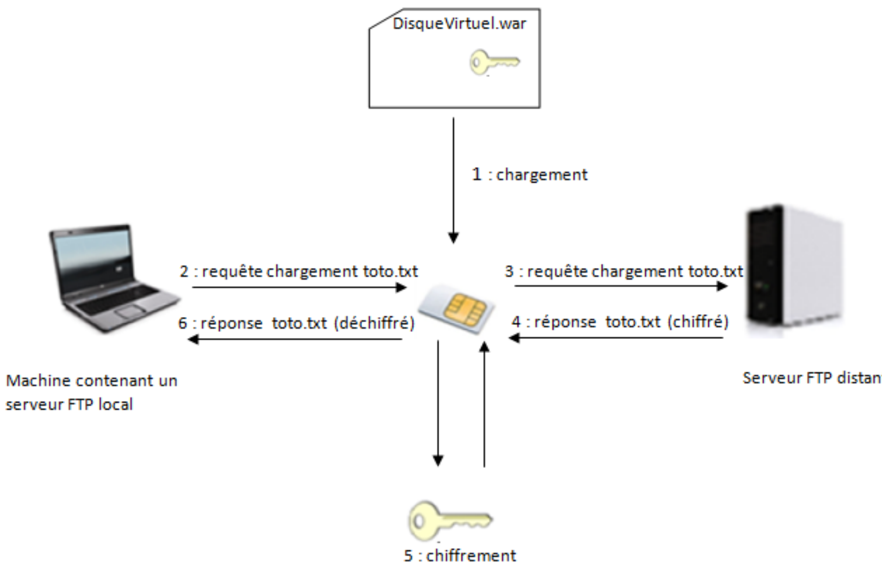

Fig. 1.8 – Exemple d'application Java Card 3 : le Disque Virtuel

alors réussir à effectuer du vol d'informations, troubler le bon fonctionnement d'un service, accéder à des ressources non autorisées, modifier des données, etc. Les mécanismes de sécurité (firewall, SSL, ..) ne suffisent pas pour protéger les applications Web. En effet, un attaquant peut s'attaquer à la logique même des applications, tout en respectant les contrôles effectués à l'extérieur du serveur Web. Les failles dans le développement des applications Web seront les premières vulnérabilités visées par l'attaquant. Ainsi, une bonne conception et de bonnes pratiques de développement de l'application Web s'imposent.

1.5.4 Exemple d'application : « le Disque Virtuel »

L'application Disque Virtuel, développée au sein de notre équipe, est un gestionnaire de fichiers sur carte à puce Java Card 3. Il permet de récupérer et d'envoyer des fichiers sur des serveurs FTP distants. Ces fichiers peuvent être chiffrés à la volée par la carte ce qui garantit une sécurité maximale en cas de compromission du serveur Web. La carte contient les clés de déchiffrement des fichiers ainsi que l'adresse où se trouvent ces fichiers. La *figure 1.8* illustre le mécanisme de transfert de fichiers via notre application.

Avant d'être chargée dans une carte à puce, notre application doit être signée avec une clé lui permettant d'avoir des permissions spécifiques : par exemple, accepter les connexions extérieures. La signature permet donc d'appartenir à un domaine de protection spécifique. Le mécanisme de transfert de fichiers est très simple. L'utilisateur doit posséder un serveur FTP sur sa machine pour pouvoir envoyer ou récupérer des fichiers. Par exemple (*figure 1.8*), pour le téléchargement d'un fichier, la carte récupère le fichier sur un serveur FTP distant, le déchiffre à la volée puis l'envoie sur le serveur FTP local. Le mécanisme de sauvegarde est simplement l'inverse du téléchargement : la carte récupère le fichier sur le serveur FTP local, le chiffre à la volée puis l'envoie sur un serveur FTP distant.

L'avantage de cette application est d'avoir un outil permettant de manipuler un grand nombre

de fichiers stockés à distance tout en garantissant leur intégrité. En cas de compromission du serveur Web où sont stockés les fichiers, un attaquant ne pourra pas les utiliser sans la clé de déchiffrement associée pour chaque fichier. On peut ainsi voir ce Disque Virtuel comme un espace de stockage où les données ne seraient visibles que par le propriétaire de la clé. L'utilisateur peut récupérer ses données à partir de n'importe quel terminal distant à condition que celui-ci dispose d'un serveur FTP local.

Notre application utilise des mécanismes de sécurité, comme une authentification par code PIN ou encore les annotations. Nous avons rendu cette application intuitive afin de montrer les capacités des cartes à puce Java Card 3 par rapport aux anciennes.

A travers cette application nous avons pu constater la facilité de développement en Java Card 3, comparé à sa précédente version Java Card 2.2. Java Card 3 se rapproche beaucoup de Java ME ce qui en fait un outil de développement plus simple à adapter pour les développeurs Java (contrairement au développement d'applications Java Card 2.2 où il est nécessaire d'avoir des connaissances spécifiques). Les transactions laborieuses à mettre en place en Java Card 2.2 ont grandement été simplifiées en Java Card 3 avec l'utilisation des annotations. De plus, contrairement à la Java Card 2.2, nous disposons d'un ramasse miette automatique pour supprimer les objets inutilisés en mémoire volatile. Le développement a aussi été simplifié par le support de nouveau type de données comme les types `Integer` et `String`.

1.6 Conclusion

La carte à puce à serveur Web embarqué constitue une grande évolution. L'ouverture de la carte aux standards du Web HTTP/HTTPS offre un accès plus simple et plus rapide aux services qu'elle offre, et une communication plus sécurisée grâce à l'utilisation du protocole TLS. Dans la plateforme Java Card 3, les mécanismes de sécurité ont été renforcés par une gestion plus fine des utilisateurs et une vérification des applications avant leur chargement (vérificateur de bytecode obligatoire) et pendant leur exécution (pare-feu, isolation de contexte d'exécution, ramasse miette, etc.). D'autre part, cette plateforme offre aux développeurs un moyen de programmation plus simple à adopter, qui se rapproche beaucoup de Java ME.

Cependant, le développement d'applications Web nécessite d'adopter des mesures afin de garantir leur sécurité et de prévenir certaines attaques. En effet, les attaques sur les applications Web sont très répandues (XSS, CSRF, etc.) et leurs conséquences peuvent être très graves. La carte manipule des données très sensibles. Une méthodologie de développement d'applications Web est nécessaire afin d'éviter qu'elles soient vulnérables aux attaques Web. De plus une mauvaise implémentation du protocole HTTP, qui ne respecterait pas la spécification, pourrait conduire à des comportements inattendus et causer des failles et des vulnérabilités.

Dans les deux parties qui suivent nous allons respectivement aborder la sécurité au niveau applicatif et au niveau du protocole HTTP implémenté.

Chapitre 2

Les attaques sur carte à puce

Sommaire

2.1 Attaques sur cartes à puce standards 26
2.1.1 Attaques physiques 26
2.1.2 Attaques logicielles 27
2.1.3 Attaques combinées 30
2.2 Les attaques Web 31
2.2.1 Exemples d'attaques Web les plus répandues 32
2.3 Conclusion 36

La carte à puce offre un environnement sécurisé pour l'exécution de plusieurs programmes et la manipulation de données. De plus, elle propose des mécanismes d'authentification très robustes. La sécurité de la carte à puce réside dans le fait qu'elle enferme des données sécurisées que le microprocesseur embarqué est en mesure de traiter en fonction des instructions fournies par le lecteur de carte. Afin de garantir la sécurité de ces opérations, plusieurs dispositifs de protection ont été apportés à la carte à puce.

Au niveau matériel, les matériaux constituant le corps de la carte visent à faire échouer les attaques chimiques d'extraction du micromodule. De plus, tous les composants sont sur le même silicium ; le microprocesseur et ses capteurs sont enrobés dans une résine, ce qui rend difficile la pose de sonde pour espionner les bus internes.

Au niveau logiciel, l'utilisation de la plateforme Java Card apporte une couche supplémentaire de sécurité. En effet, l'utilisation d'une machine virtuelle apporte un contrôle supplémentaire en empêchant l'accès direct aux registres du processeur. De plus, des mécanismes de sécurité des applications ont été mis en place, tels que le pare-feu (*firewall*), la vérification de type et l'impossibilité de construire des pointeurs qui pourraient être utilisés dans des programmes malicieux afin d'extraire le contenu de la mémoire. Ces mécanismes empêchent qu'une application hostile crée des dommages à d'autres applications et au système.

Malgré tous ces mécanismes de sécurité, des attaquants peuvent passer outre ces dispositifs de protection et avoir un accès direct à la carte. De plus, une carte Java est dite ouverte, c.à.d des applications peuvent y être chargées et exécutées après délivrance (post-usurance). Des attaquants

peuvent donc exploiter des failles d'implémentation algorithmiques pour introduire des applications qui comportent du code malicieux.

2.1 Attaques sur cartes à puce standards

Il existe deux types principaux d'attaques sur les cartes à puce, les attaques physiques ou matérielles qui se font sur la carte en tant que composant matériel électronique, et les attaques logiques qui exploitent des failles algorithmiques ou des défauts dans les mécanismes d'isolation (dans la Java Card).

2.1.1 Attaques physiques

Ce premier type d'attaque suppose une connaissance approfondie dans le domaine de l'électronique et de l'architecture des cartes à puce. Elle se divise en deux catégories : les attaques invasives et les attaques non invasives.

Attaques invasives : elles nécessitent de retirer physiquement ou chimiquement le circuit électronique de la carte en plastique, ce qui entraîne la destruction de la carte, contrairement aux attaques non-invasives qui n'altèrent pas son utilisation. Ces attaques sont en général menées par des experts et requièrent du matériel très couteux (par exemple une sonde ionique focalisée). Elles visent à récupérer le maximum d'informations sur le circuit pour en déduire les mécanismes de sécurité mis en place.

Les attaques non-invasives : contrairement aux attaques vues précédemment, ce type d'attaques vise à observer ou perturber certaines fonctionnalités physiques de la carte à puce sans altérer son fonctionnement. La carte est alors réutilisable après l'attaque. Nous présentons dans ce qui suit des exemples d'attaques non-invasives.

- *Attaques par conditions anormales :* ces attaques consistent à perturber le fonctionnement normal de la carte dans le but de modifier son comportement. Elles peuvent s'effectuer en perturbant les entrées de la carte comme la tension ou la fréquence d'alimentation. Ces perturbations peuvent aussi être externes, en altérant par exemple la température. Toutefois, les cartes à puce sont généralement dotées de détecteurs de conditions anormales qui rendent ce type d'attaque difficile à réaliser.
- *Attaque par canaux cachés :* ce type d'attaque consiste à mesurer un paramètre physique extérieur pendant l'activité de la puce. Ce paramètre peut être le temps de calcul (*Timing Analysis*), le courant (DPA : *Differential Power Analysis*, SPA : *Simple Power Analysis*) [SCDC+11, RO04] ou le champ magnétique émis par la puce (SEMA : *Simple Electro-Magnetic Analysis*, DEMA : *Differentiel Electro-Magnetic Analysis*)[Qui11]. L'attaquant mesure alors les variations des paramètres physiques. Après analyse des données relevées, il pourra déduire des informations sur l'exécution du logiciel sur la carte, pouvant aller jusqu'à la divulgation des données de sécurité (clés cryptographiques ou données d'authentification). En effet, les consommations de courant ou de radiation de chaque instruction sont différentes par donnée manipulée.

– *Attaque par injection de fautes :* c'est l'une des attaques physiques les plus répandues. Elle s'appuie sur une propriété du silicium, qui dans certaines conditions, change de comportement électrique. L'attaque consiste donc à exploiter cette propriété pour perturber l'exécution des programmes embarqués dans la puce, tels que les algorithmes cryptographiques, dans le but de les pousser à avoir un comportement inhabituel qui pourra être exploité [Gir07]. Il est possible de perturber le fonctionnement d'un circuit électronique avec des rayonnements tels que les ultraviolets, rayons EM, rayons X, lumière blanche, etc. Le but est de modifier le contexte d'exécution des applications embarquées ou d'altérer une partie du contenu de la mémoire. L'attaque en faute peut avoir deux conséquences différentes : elle peut être soit permanente de telle sorte que la valeur d'une cellule mémoire soit définitivement changée ; soit transitoire modifiant temporairement une opération et/ou la valeur d'une variable lors de son transit sur le bus de données.

Toutefois, la présence de détecteurs embarqués dans la puce permet de limiter ces attaques. Les cartes à puce sont souvent dotées de contre-mesures adaptées à ces attaques, consistant en particulier à insérer des délais aléatoires dans l'exécution des programmes, de manière à les rendre moins facilement observables. Certaines cartes bancaires possèdent des détecteurs d'illuminations laser.

La spécification Java Card 3 est considérablement plus complexe que les précédentes, et l'observation des programmes pourrait s'en trouver compliquée, en particulier, en raison de la présence de multiples fils d'exécution (multi-tâches).

2.1.2 Attaques logicielles

Elles consistent à s'attaquer à la partie logicielle de la plateforme par injection de données ou chargement d'applications malicieuses dans le but de contourner l'exécution des applications installées dans la carte à puce ou de dévoiler ses secrets (clés, code Pin, etc.). Elles englobent deux types d'attaque : les attaques qui exploitent les failles algorithmiques dues soit au non-respect de la spécification ou à des failles non abordées dans les spécifications elles mêmes, et les attaques par chargement d'applications malicieuses dans la carte à puce.

Direct protocol attaks et le *Fuzzing* sont deux techniques d'attaques présentées dans [joi09] basées sur la manipulation des commandes de communication entre terminal et carte à puce. Le *fuzzing* consiste à envoyer une suite de tests dans le but d'obtenir des réponses inattendues retournées par la carte. Le *fuzzing* permet à l'attaquant de détecter des vulnérabilités ou des bogues qui peuvent être exploités par la suite. L'attaque *Direct protocol attacks* : consiste à envoyer des commandes auxquelles la carte à puce ne s'attend pas dans son état courant. Toutefois, de telles attaques ne sont possibles que si le protocole d'échanges entre la carte et le terminal est connu. Avec Java Card 3.0, ces attaques demeurent possibles au niveau des échanges de commandes cartes. Avec l'édition connectée, elles le sont également au niveau des protocoles réseau, basés sur TCP/IP et HTTP, qui ouvrent de nouvelles voies d'attaques (attaques Web), détaillées plus loin.

Pour garantir les règles de sécurité (c'est-à-dire intégrité, confidentialité et disponibilité), des mécanismes ont été définis dans les spécifications Java Card. En effet, le vérificateur de bytecode (bytecode verifier ou BCV) vérifie qu'une applet est sémantiquement correcte et le pare-feu empêche qu'une applet accède ou modifie une autre applet ou une autre ressource qui n'appartient pas à

son contexte. Corependant,il existe un mécanisme permettant l'accès à partir d'une applet à une référence d'une instance définie comme partageable (shareable) appartenant à une autre applet.

D'autre part les cartes Java Card étant des cartes ouvertes, elles permettent de charger et d'ajouter de nouvelles applications après leur émission. Elles sont donc sensibles à des attaques logicielles qui tentent d'introduire des applications qui contournent les mécanismes de sécurité mis en place. Dans les versions de Java Card précédant la version 3.0, le vérificateur de bytecode n'est pas obligatoire dans la carte ; il est donc possible de charger une application sémantiquement incorrecte dans le but de réaliser des attaques. Cependant, des travaux [BDH11a, BICL11, BL12] ont montré qu'il est possible outrepasser la vérification de bytecode en utilisant une combinaison d'attaques physique et logique. Dans la suite, nous présentons quelques exemples d'attaques logicielles pures.

Attaque par confusion de type. Une des briques de sécurité de la Java Card repose sur la sûreté du typage [ICL10], normalement assurée par le vérificateur du bytecode avant la transformation d'une applet en fichier CAP. Cependant, dans le cas où le chargement des applications sur la carte ne se fait pas uniquement par ses fabricants (exemple : un tiece de confiance), le fichier CAP peut être altéré avant son chargement dans la carte.

Dans une attaque par confusion de type, présentée par Mostowski et al. [MP], un tableau d'octets est transformé en un tableau d'entiers courts permettant ainsi l'accès à une zone de mémoire plus grande (pouvant appartenir à d'autres applets par exemple). Pour cela, l'attaque abuse de mécanisme de partage d'interface. Pour mieux comprendre cette attaque, supposons deux applets : une applet serveur qui expose une interface de partage, et une applet client qui accède à cette interface. Le serveur envoie la référence de son tableau d'octets à l'applet client via la méthode `giveArray()`, l'applet client la renvoie ensuite via l'interface mal typée (`accessArray()`) retournant ainsi un tableau d'entiers courts pour un tableau d'octets.

Listing 2.1– Interface de l'applet client

```
1 public interface Interface extends Shareable {
2 public byte [ ] giveArray ( ) ;
3 public short accessArray(byte [ ] MyArray) ;
4 }
```

Listing 2.2– Interface de l'applet serveur

```
1  public interface Interface extends Shareable {
2      public byte [ ] giveArray ( ) ;
3      public short accessArray(short [ ] MyArray) ;
4 }
```

Cette même attaque peut été réalisée d'une autre façon en exploitant l'instruction NOP définie dans la spécification Java Card comme n'effectuant aucune action. Elle consiste à charger un programme auto modifiable dans la carte permettant de lire ou d'écrire n'importe où dans la mémoire de la carte. Il est ainsi possible de changer certaines instructions par l'instruction NOP. Dans

l'exemple présenté dans *listing 2.3*, les instructions des lignes 7 à 10 du code binaire (*listing 2.4*) sont remplacées par des instructions NOP. Ainsi, la dernière valeur empilée devient la référence vers le tableau `array` (instruction `aload_1`). La méthode retournera donc une réference vers un tableau d'entiers courts.

Listing 2.3– Exemple d'attaque par instruction NOP

```
1  public short getMyAddresstabByte (byte [ ] array ) {
2     short foo =(byte ) 0x55AA ;
3     array [ 0 ] = (byte ) 0xFF ;
4     return foo ;
5  }
```

Listing 2.4– Code binaire

```
1  public short getMyAddressTabByte (byte [ ] array ) {
2  03 // flags : 0 max_stack : 3
3  21 // nargs : 2 max_locals : 1
4  10 AAbspush 0xAA
5  31 sstore_2
6  19 aload_1
7  03 sconst_0
8  02 sconst_m1
9  39 sastore
10 1E sload_2
11 78 sreturn
12 }
```

Attaque par abus des instructions manipulant des membres statiques. Afin d'optimiser la phase d'édition des liens, des symboles sont utilisés dans le fichier CAP pour le référencement des méthodes ou des champs. Ces symboles sont listés dans le composant *Constant Pool* et les adresses relatives à chaque symbole à résoudre sont contenues dans le composant *Reference Location*. L'édition des liens consiste à vérifier que chaque opérande avec un symbole défini dans le composant *Constant Pool* pointe vers un élément existant avec un type attendu par l'opérande.

Pour des raisons de capacité mémoire et de gain de performance, il est possible que le pare-feu ne contrôle pas les accès aux éléments statiques (`getStatic`, `putStatic` et `invokeStatic`). Cette exception au principe d'isolation du pare-feu provient du fait que certains objets statiques du système (tampon APDU, mécanisme d'exception, etc.) doivent être accessibles dans tous les contextes.

En absence du vérificateur bytecode, cette propriété peut être exploitée par un attaquant pour modifier des références sur des éléments statiques afin d'accéder à des ressources appartenant à un contexte de sécurité différent de celui de l'applet courante. Tiana et al. proposent une librairie Java, le CapMap [RBL12], permettant de déréférencer dans le composant *Reference Location*, le

paramètre d'une instruction `getStatic` ; ce paramètre est ensuite utilisé pour lire des parties de la mémoire non accessible dans un usage normal.

Réalisation d'un cheval de Troie dans une carte (EMAN). EMAN est un ensemble d'attaques réalisées dans l'équipe SSD de l'université de Limoges. Elle exploite l'attaque précédente afin d'utiliser les instructions `getStatic` et `putStatic` pour parcourir la mémoire et modifier des applets installées. L'attaque EMAN1 [ICL10] consiste à réaliser un cheval de Troie en utilisant des applets malicieuses qui peuvent inclure des instructions ne respectant pas les règles de typage de Java. Cette attaque a pour but de modifier le flux d'exécution d'une applet présente dans la carte, en recherchant un motif dans la mémoire de la carte et de le modifier. Grâce à cette attaque, il est, par exemple, possible de supprimer la vérification du code PIN dans une applet, en remplaçant dans la mémoire, le code associé à l'instruction qui génère une exception en cas de code PIN non-valide par le code de l'instruction NOP.

L'attaque EMAN2 [BICL11] est basée sur le même principe que l'attaque précédente. Son objectif est de renvoyer le pointeur d'exécution à une adresse bien définie, et de modifier le registre contenant les adresses de retour des méthodes par l'adresse d'un tableau contenant un code malicieux.

2.1.3 Attaques combinées

Sur la catégorie des attaques précédentes, la spécification Java Card 3.0 change considérablement la donne. Grâce aux nouveaux mécanismes de sécurité disponibles sur l'édition connectée, en particulier la vérification de bytecode qui est pratiquée systématiquement lors du chargement des programmes sur la plateforme et le ramasse-miette qui est automatique. Les attaques logiques visant spécifiquement Java Card deviennent plus difficiles à mettre en œuvre. Cependant pour réussir à contourner le vérificateur de bytecode, un attaquant peut faire une combinaison d'attaques matérielle et logicielle. Cette combinaison consiste à charger dans la carte des applications sémantiquement correctes qui sont par la suite modifiées par une attaque physique pour générer un code malicieux permettant de réaliser des attaques logiques. L'application modifiée est appelée mutant.

La génération de mutants est un nouvel axe de recherche [BDH11a, BICL11, BL12]. Nous présentons quelques exemples de cette catégorie d'attaques dans ce qui suit.

Perturbation de l'instruction checkcast. Barbu et al. proposent [BDH11a] une attaque par perturbation de l'instruction `checkcast` permettant d'utiliser une arithmétique de pointeur sur une plateforme Java Card. L'attaque commence par créer une confusion de type. L'instruction checkcast de la JCVM est utilisée dans le but de vérifier la validité d'une conversion de type. Les auteurs appliquent une attaque par injection de faute (attaque physique) sur cette instruction dans le but de générer une conversion incorrecte qui lancerait une exception à chaque exécution. Grâce à cette exception les auteurs sont parvenus à synchroniser l'activation du faisceau laser et le fetch du code natif conditionnant le succès ou l'échec de la conversion de type.

Une fois la perturbation réussie, l'attaquant peut donc accéder à un même objet x de type X par un champs instance d'une classe différente X ou Y. Il est ainsi possible d'utiliser une arithmétique de pointeur comme dans le langage C ce que le langage Java proscrit formellement.

Perturbation des données contenues dans la pile d'opérandes. La machine virtuelle Java définit une pile d'opérandes où sont sauvegardés les paramètres et les retours des instructions composant un bytecode. Par exemple, l'exécution de l'instruction `iadd` opérant l'addition de deux entiers de type `int`, consiste à retirer deux éléments au sommet de la pile des opérandes, puis à effectuer une addition sur ces deux valeurs dont le résultat de l'addition est ajouté au sommet de la pile des opérandes.

Contrairement à l'attaque précédente qui utilise l'attaque en faute pour perturber une instruction, les auteurs dans [BDH11b] s'intéressent à la perturbation des données manipulées. Ces données sont perturbées par une attaque physique lors de l'empilement des opérandes dans la pile. Les auteurs proposent principalement trois attaques. La première consiste à injecter une faute lors de l'empilement d'une variable booléenne précédant l'exécution d'un saut conditionnel (c'est à dire un bloc if en langage Java) afin d'influencer la prochaine instruction à exécuter. La seconde attaque permet de réaliser une confusion de type avec une forte probabilité en combinant cette attaque physique avec une application malicieuse créant une multitude d'instances d'une classe donnée. Enfin, la dernière attaque montre d'une manière similaire comment créer une confusion entre deux instances d'une même classe, ou plus précisément entre deux instances de deux classes implémentant une même interface.

2.2 Les attaques Web

Les débuts du World Wide Web (ou WWW) furent relativement statiques. Le serveur Web avait pour seule fonction de fournir des fichiers statiques comme des pages Web, écrites en HTML. Au fil du temps, les développeurs ont intégré l'accès à un contenu dynamique (JavaScript, AJAX, etc.). Depuis, les applications Web sont devenues un moyen dominant d'accès à des services en ligne. Le nombre et l'importance des applications Web augmentent rapidement, mais au même temps, l'impact des vulnérabilités devient de plus en plus important.

Il existe de nombreuses attaques sur les services Web ; des organisations [OWAa, Mit, WAS] tentent d'effectuer un classement de ses attaques selon les plus répandues et les plus dangereuses, mais cette tache reste difficile à réaliser vu que beaucoup de sites évitent de dévoiler qu'ils en ont été victimes.

L'intégration du Web au monde de la carte ouvre de nouvelles portes à des attaques déjà existantes sur le Web standard [KICL10]. Ces différentes attaques sont généralement liées à des vulnérabilités des applications. Elles peuvent provoquer des comportements indésirables tels que l'indisponibilité partielle des services, la mise en péril des données confidentielles, voir même la perte de contrôle au profit d'un utilisateur malveillant. Les applications Web pour carte à puce ne sont pas concernées par toutes les vulnérabilités Web standards. Par exemple : l'attaque par injection de code SQL peut être écartée, vu que les bases de données et le langage SQL ne sont pas supportés dans les cartes à puce. Cependant, d'autres attaques restent possibles, notamment les attaques par injection de code (XSS) qui sont très puissantes et très répandues.

La meilleure façon de se prémunir des attaques Web est de tenir compte de la sécurité depuis les phases de conception et de développement des applications Web en appliquant quelques bonnes pratiques.

2.2.1 Exemples d'attaques Web les plus répandues

Dans cette section nous citons quelques exemples d'attaques les plus répandues, identifiées par l'organisation OWASP [OWAa] qui a sélectionné les 10 attaques les plus répandues, recensées en 2010 (tableau 2.1). Nous précisons également dans quelle mesure certaines d'entre elles ne peuvent pas se produire sur les cartes à puce et nous proposons quelques recommandations de développement pour se prémunir de chaque attaque.

Classement	**Risques**
A1	Injection de commandes
A2	Failles Cross Site Scripting (XSS)
A3	Violation de gestion d'authentification et de session
A4	Référence directe à un objet non sécurisée
A5	Cross Site Request Forgery (CSRF)
A6	Configuration non sécurisée
A7	Stockage non sécurisé
A8	Manque de restriction d'accès URL
A9	Communications non sécurisées
A10	Redirection et renvoi non validés

Table 2.1 – Sélection d'OWASP des dix vulnérabilités les plus répandues en 2010

1. **Injection de commandes**

   L'attaque par injection consiste à injecter du code malicieux dans des entrées présentées à l'utilisateur [KGJE09]. Ce code malicieux est ensuite interprété par le serveur comme des instructions qui peuvent consister en des requêtes d'interrogation de base de données (SQL), des commandes système, ou des commandes d'accès à des fichiers. Cette attaque concerne moins la carte à puce dont l'accès est beaucoup plus réduit qu'un dispositif standard (ordinateur) ; de plus, pour des contraintes de ressources, le stockage des fichiers, les bases de données et le langage d'interrogation de base de données ne sont pas prévus dans la carte à puce.

   Cependant pour se prémunir de cette attaque, le développeur doit filtrer toutes les données entrées par l'utilisateur pour empêcher que certains caractères qui peuvent avoir un autre sens que celui attendu puissent être interprétés dans le serveur ou le système.

2. **Cross Site Scripting (XSS)**

   XSS est l'une des attaques les plus répandues sur le Web. Elle peut être considérée comme une attaque par injection [HTL+05], sauf que dans ce cas le code malicieux est interprété au niveau du client (navigateur Web). Elle exploite des failles dans des applications Web qui manipulent des données externes à l'application sans les filtrer au préalable. Elle consiste à injecter du code malicieux (des scripts souvent écrits en JavaScript) dans la page Web d'une application vulnérable. Si l'application retourne une ressource contenant ce code malicieux alors ce dernier sera exécuté au niveau du client (navigateur Web).

   Le but de cette attaque est généralement de récupérer les identifiants de session, [Gar09], rediriger l'utilisateur vers un domaine qui est sous le contrôle de l'attaquant, causer un dénis de service ou modifier les pages Web de l'application.

La meilleure prévention contre ce type d'attaque consiste principalement à prévoir un encodage des données en sortie qui doivent être converties en leur encodage HTML équivalent afin de garantir que le navigateur Web ne traite pas certains caractères, potentiellement malicieux, comme une partie de la structure du document HTML mais plutôt comme une partie de son contenu [OWAa]. Nous nous intéressons particulièrement à cette attaque qui fera l'objet du chapitre suivant.

3. **Violation de gestion d'authentification et de session**
   L'authentification joue un rôle important dans la sécurité des applications Web. Elle consiste à allouer des privilèges particuliers d'accès à des services Web, selon l'identité de l'utilisateur. La gestion des sessions des utilisateurs authentifiés à une application consiste à garder en mémoire des informations relatives à l'utilisateur connecté (par le biais des cookies).

   Les attaques sur la violation de session consistent à : utiliser la force brute en testant un ensemble d'identifiants et mots de passe jusqu'à arriver aux valeurs acceptées [SWS02], intercepter les données de session enregistrées en mémoire (les cookies) en utilisant une attaque XSS par exemple ou encore en forgeant un jeton de session valide.

   Ces attaques exploitent des faiblesses dans les mécanismes d'authentification, souvent introduites par des fonctions auxiliaires d'authentification, telles que la déconnexion, la gestion de mots de passe, les questions secrètes, les mises à jour des informations relatives aux comptes, les messages d'erreurs.

   Pour éviter le risque de violation de sessions il est nécessaire de refuser tout identifiant de session reçu à partir d'une URL afin de se prémunir des attaques par fixation de session [4] [JBSP11]. Il faut également commencer le processus d'ouverture de session par une page chiffrée et un nouveau jeton de session. Le jeton de session doit être regénéré à chaque nouvelle authentification. Il est également recommandé de mettre une limite à la durée de validité d'une session au-delà de laquelle une déconnexion automatique est effectuée si l'utilisateur est inactif.

4. **Référence non sécurisée à un objet**
   Il est dangereux d'afficher aux utilisateurs, via une URL ou un paramètre dans un formulaire, des références à des objets internes tels qu'un fichier, un répertoire, une clé, etc. Un attaquant peut manipuler le paramètre dans le but de violer la politique de contrôle d'accès mise en place et d'accéder à des ressources non autorisées. Dans l'exemple suivant si le paramètre `UserID` est passé en paramètre d'une URL alors il suffit de le changer pour récupérer le numéro de compte `cartID` de n'importe quel utilisateur.

Listing 2.5– référence non sécurisée à cartID

```
1  int cartID = Integer.parseInt(request.getParameter("UserID"));
2  String query = getcartID(UserID);
3  out.println("your cartID is"+ cartID);
```

4. L'attaque par fixation de session exploite des vulnérabilités d'applications permettant à un utilisateur de fixer l'identifiant de session d'un autre utilisateur

La meilleure protection contre cette attaque est de ne jamais exposer de références directes aux utilisateurs vers des implémentations internes d'objets. Il faut utiliser des index ou des équivalences par référence indirecte. Pour plus de sécurité, la méthode d'acceptation des bonnes valeurs *Accept known good* permet de vérifier que les données appartiennent à un ensemble de valeurs valides qui peuvent être définies par des critères comme : le type, la longueur, la syntaxe, etc. Toutes les données qui ne correspondent pas à ces critères devraient être rejetées. Cependant, si une référence directe à un objet doit absolument être utilisée, il faut demander à l'utilisateur de ressaisir son identifiant et mot de passe avant de pouvoir y accéder.

5. **Cross Site Request Forgeries (CSRF)**

Cross-Site Request Forgery (CSRF) est une attaque très proche de l'attaque XSS [Gol08] car elle consiste également à inciter une victime à charger une page qui contient une requête malicieuse. Son objectif est d'utiliser l'identité et les privilèges d'une victime pour effectuer une fonction indésirable au nom de cette victime [BGH+12]. Autrement dit ce type de vulnérabilité peut potentiellement permettre de réaliser une action non autorisée dans une application Web avec le droit d'un utilisateur légitime et ce, sans son consentement.

L'attaque est réalisée quand un utilisateur authentifié à un site (site d'une banque par exemple) clique sur un lien malicieux envoyé par l'attaquant, qui masque dans ce lien une requête vers l'application à laquelle l'utilisateur est authentifié (transfert d'argent vers le compte de l'attaquant par exemple). Le lien malicieux peut être marqué dans des balises d'images, et l'attaquant doit inciter l'utilisateur authentifié sur le site *bank.com* à consulter ce lien, en lui faisant croire qu'il s'agit d'une image.

```
<img src="http://bank.com/transfer.do?acct=pirate&amount=100000"
width="1" height="1" border="0">
```

Quand l'utilisateur clique sur ce lien, une petite fenêtre s'affiche indiquant que l'image n'a pas pu être affichée et la requête de transfert d'argent est envoyée au serveur sans que l'utilisateur ne s'en aperçoive.

Cette attaque peut avoir plusieurs conséquences, telles que : le changement de mot de passe, l'achat en ligne, l'achat d'action sur un site boursier, l'envoi d'e-mails, etc.

Les sites qui implémentent des requêtes POST sont moins exposés. En effet, les requêtes GET sont facilement prédictibles car les informations transitent dans l'URL ; il suffit donc de forger une URL afin d'envoyer une requête valide au serveur Web [OWAa].

L'utilisation d'un jeton (token) est un moyen de protéger la session de l'utilisateur. Le jeton permet de vérifier la validité de l'origine d'une requête. Le jeton doit être généré aléatoirement pour éviter qu'il soit prédictible par un attaquant et placé de façon cachée dans le formulaire pour que l'utilisateur ne détermine pas son existence. Il est régénéré à chaque requête et si le jeton envoyé par le navigateur est différent de celui stocké dans l'application alors la requête est rejetée. Pour plus de sécurité, une durée de vie peut être ajoutée au jeton de session, pour déterminer un temps au-delà duquel il sera expiré. Une durée minimale aussi, suppose que la saisie a été faite par un attaquant automatique (robot).

6. **Configuration non sécurisée**

Cette faille regroupe l'ensemble des vulnérabilités qui peuvent être dévoilées par l'architecture des serveurs ou des applications. Le moyen le plus répandu est les messages d'erreur.

Les messages d'erreurs peuvent dévoiler d'importants détails sur l'implémentation de l'application ou du système qui permettraient à un attaquant de détecter et d'exploiter des vulnérabilités. Un exemple simple est celui de l'authentification. Si une application génère des messages d'erreur qui spécifient si c'est le mot de passe ou l'identifiant qui est incorrect alors une attaque par force brute pour détecter l'identifiant et le mot de passe est simplifiée. Si le message d'erreur indique que le mot de passe est incorrect, alors l'attaquant en déduit que l'identifiant qu'il a forgé est correcte et il va donc le fixer et continuer son attaque sur uniquement le champ associé au mot de passe.

Pour prévenir cette faille, une discipline dans la génération des erreurs est indispensable. Il ne faut jamais afficher à l'utilisateur des messages d'erreur détaillés contenant des informations de débogage ou des informations relatives aux chemins. Il est important de créer un gestionnaire d'erreurs par défaut qui retourne à l'utilisateur des messages d'erreur correctement nettoyés. Et pour éviter qu'un attaquant déduise des informations à partir d'une estimation du temps de réponse de chaque opération, il serait utile d'appliquer un temps de réponse aléatoire pour toutes les transactions.

7. **Stockage de données non sécurisé**

   Une faille de stockage de données non sécurisées existe si une application Web ne protège pas correctement les données sensibles, telles que les numéros de cartes de crédit, de sécurité sociale, les informations d'authentification, avec un algorithme de chiffrement ou de hash approprié. Dans ce cas, un attaquant peut les récupérer ou les modifier.

   Cette attaque concerne moins les cartes à puce qui utilisent des algorithmes de chiffrement avec des clés de chiffrement bien protégées (aspect "tamper-resistent" de la carte). La recommandation que nous pourrions faire dans ce cas est d'utiliser uniquement les algorithmes de chiffrement présents dans la carte.

8. **Défaillance dans la restriction des accès URL**

   Un attaquant peut tenter d'accéder par une URL à des ressources cachées non accessibles au public ou à des répertoires contenant des informations sensibles. Un pirate peut s'y prendre par force brute en utilisant en général des conventions de nommage communes et des emplacements standards pour la plupart des applications.

   L'accès à ces fichiers permet à l'attaquant d'accéder à des informations sensibles pour une application Web, et d'apporter des modifications dans le but d'avoir des droits d'accès ou des autorisations pour certaines transactions sensibles, etc. Il arrive que les développeurs limitent leur protection à la couche présentation, en cachant aux utilisateurs non autorisés, des URL et des liens associés à certaines pages réservées à des utilisateurs particuliers. Cette protection ne suffit pas. Si un attaquant intercepte ou découvre l'existence de cette URL, il pourra accéder à des données ou fonctions auxquelles il n'est pas autorisé.

   Les applications Web doivent renforcer le contrôle d'accès à chaque URL et chaque fonction métier et bloquer les accès à tous les fichiers que l'application n'utilise pas. Il est nécessaire d'appliquer un filtre autorisant uniquement les types de fichiers qui peuvent être utilisés, par exemple fichiers HTML, et bloquer les différents essais d'accès à des fichiers d'événements (logs), aux fichiers XML, etc.

L'application du mécanisme de contrôle d'accès aux URL et aux fonctions métiers consiste à vérifier à chaque étape, les rôles et les droits associés aux utilisateurs à chaque traitement.

Cette attaque s'applique aux cartes à puce à serveur Web embarqué uniquement dans certaines mesures. En effet, les serveurs embarqués sont simples et ne peuvent pas contenir autant de données qu'un serveur standard. De plus, il n'existe pas de console dans la carte à puce, il n'existe donc pas un moyen d'accéder au système. Par conséquent, des URLs cachées seraient peu probables. Cependant, une administration à distance des applications étant possible, alors des failles pourraient être exploitées à ce niveau.

9. **Communication non sécurisée**
Un attaquant peut écouter le trafic d'un réseau dans l'objectif d'intercepter les informations qui y transitent [SWS02], notamment : les éléments d'authentification et les données sensibles échangées. Cette attaque est généralement utilisée pour pouvoir collecter des informations permettant d'effectuer d'autres attaques plus néfastes.

Pour éviter ce genre de vulnérabilités, l'application doit chiffrer correctement les communications authentifiées et sensibles. Il est nécessaire d'utiliser TLS sur toutes les connexions authentifiées et durant la transmission des données sensibles, telles que les données d'authentification, numéro de carte bancaire, etc.

10. **Redirection et renvoi non validés**
Une application Web peut contenir des redirections vers les pages Web qui la composent. Souvent ces redirections incluent des paramètres fournis par l'utilisateur dans l'URL de destination. Si ces paramètres ne sont pas validés, alors l'application est vulnérable à une attaque pouvant rediriger la victime vers un autre site Web.

Cette attaque est utilisée généralement dans le but de rediriger la victime vers un site malveillant sous le contrôle de l'attaquant ou passer outre les contrôles de sécurité et accéder à des données ou des fonctions non autorisées.

Pour se prémunir de cette attaque, il faut configurer la redirection de telle sorte qu'elle se limite uniquement à des pages locales à l'application. Il faut également éviter d'utiliser les paramètres parvenant d'un utilisateur dans la définition des URLs ciblées. Si utiliser des paramètres utilisateurs est nécessaire alors il est recommandé de valider chaque paramètre pour vérifier qu'il est autorisé en utilisant une table de correspondance entre les paramètres utilisateurs et les pages à autoriser.

2.3 Conclusion

Dans ce chapitre nous avons présenté les différents types d'attaques possibles sur la carte à puce. Avec l'intégration d'un serveur Web, la carte à puce s'ouvre à un nouveau volet d'attaques : les attaques Web. La plupart des attaques Web que nous avons présentées sont causées par des données malicieuses introduites dans des champs ou des paramètres accessibles à l'utilisateur. Les attaques par injection de code et particulièrement les XSS, sont les plus répandues sur le Web, elles exploitent des failles dans des applications qui ne vérifient pas les entrées provenant d'un utilisateur et ne filtrent pas les sorties de l'application qui sont retournées vers le client (navigateur). L'application

devient dans ce cas un point de relais d'attaques, ce qui peut mettre en doute la confiance de l'utilisateur en sa carte à puce.

Les XSS sont relativement faciles à réaliser mais leurs conséquences peuvent être très graves (récupération d'informations, modification de la logique de l'application, dénis de service). Nous nous sommes particulièrement intéressés à ce type d'attaque que nous présentons plus en détail dans le chapitre suivant.

Deuxième partie

Sécurité des applications Web Java Card 3

Chapitre 3

Les attaques Cross Site Scripting

Sommaire

3.1 Description **41**
3.1.1 Injection volatile 41
3.1.2 Injection persistante 43
3.2 Portée et exploitation des attaques XSS **44**
3.3 Solutions existantes **45**
3.3.1 Solutions côté client 46
3.3.2 Solutions côté serveur 47
3.4 Conclusion **52**

3.1 Description

La *Cross Site Scripting* (appelé aussi XSS) consiste à injecter du code malicieux dans une entrée d'une application Web et à faire interpréter cette entrée dans le navigateur Internet d'une victime [Kle]. Le but de cette attaque est généralement de transférer des données à une tierce personne (un attaquant), de provoquer un comportement inattendu de l'application Web ou de causer une indisponibilité de service. Une XSS peut cibler n'importe quelle application Web indépendamment du langage de programmation utilisé (Java, PHP, . . .). Elle exploite des sites Web vulnérables qui manipulent ou affichent dynamiquement du contenu utilisateur (du code HTML ou des scripts) sans effectuer de contrôle et d'encodage de ce contenu au préalable.

Il existe en général deux façons d'injecter du code malveillant dans une page Web affichée à l'utilisateur [KGJE09]. D'une manière persistante (XSS persistante) ou non persistante (XSS volatile).

3.1.1 Injection volatile

Elle consiste à injecter des données malicieuses dans un lien qui sera envoyé à la victime. Quand la victime clique sur ce lien, le code injecté s'exécute sur son navigateur Web. Pour illustrer cette attaque, supposons une application Web `vulnerableSite.net` qui fournit une page Web contenant

un champ « nom » où l'utilisateur pourra introduire son nom d'utilisateur : nom_de_la_victime. L'envoi de cette donnée au serveur correspond à l'URL suivante :

```
http://www.vulnerableSite.net/?nom=nom_de_la_victime
```

Si l'application retourne dynamiquement un message qui contient les données entrées par l'utilisateur, par exemple : «Bonjour nom_de_la_victime», sans les avoir vérifiées et filtrées au préalable, alors l'application est vulnérable à une XSS volatile. Pour vérifier la présence de cette vulnérabilité il suffit de remplacer dans l'URL, «nom_de_la_victime» par le script `<script>alert('XSS')</script>`. La *figure 3.1* montre que ce script a bien été interprété et une fenêtre de dialogue s'affiche.

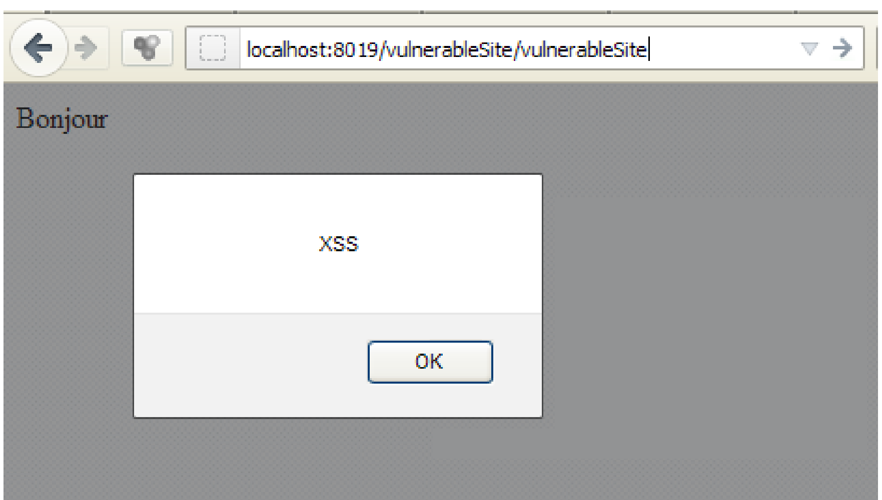

Fig. 3.1 – Affichage d'une fenêtre de dialogue par un script

Maintenant que nous avons détecté une vulnérabilité dans l'application, nous pouvons l'exploiter en injectant un code plus malicieux pour récupérer des informations sensibles. L'interprétation du code JavaScript injecté dans le lien ci-dessous permet de récupérer les cookies de la victime.

```
http://www.vulnerableSite.net/?nom=
<SCRIPT> document.location='http://site.pirate/cgibin/
script.cgi?'+document.cookie </SCRIPT>
```

Pour faire exécuter ce script, l'attaquant doit convaincre l'utilisateur de cliquer sur ce lien en utilisant n'importe quel moyen de *phishing* [5] ou d'ingénierie sociale [6]. Quand l'utilisateur clique sur ce lien, le code JavaScript s'exécute dans son navigateur Web et ses cookies de session sont envoyés au site *site.pirate*. L'attaquant peut désormais utiliser les cookies récupérés pour effectuer d'autres attaques ou récupérer d'autres données, en usurpant l'identité de la victime.

Cependant, présentée de cette manière, l'URL peut facilement être détectée comme douteuse. Pour cela, il existe différents moyens permettant de masquer l'attaque, telle que l'encodage ou le raccourcissement de l'URL.

5. Technique qui consiste à faire croire à une victime qu'elle s'adresse à un tierce de confiance afin de lui soutirer des renseignements personnels.
6. Pratique qui consiste à exploiter les failles humaines et sociales de la victime pour obtenir des informations.

3.1.2 Injection persistante

Elle ne représente que 25% des attaques XSS, mais elle est beaucoup plus puissante que les attaques XSS volatiles. L'attaquant enregistre de manière permanente le code malicieux dans une ressource gérée par l'application Web, telle qu'un fichier ou une base de données. L'attaque se réalise ultérieurement, quand une victime demande l'accès à une page dynamique qui accède à cette ressource. Nous avons réalisé cette attaque sur notre application Java Card 3 : «Disque virtuel». Nous présentons dans ce qui suit comment cette attaque a réussi à rendre l'application presque inutilisable.

Exemple d'attaque XSS sur l'application Java Card 3 « Disque Virtuel »

L'application « Disque Virtuel » présentée dans le chapitre précédent permet d'accéder à des fichiers stockés sur des serveurs FTP distants. Ces fichiers peuvent être chiffrés à la volée par la carte, ce qui garantit une sécurité maximale en cas de compromission du serveur Web. La carte contient les clés de déchiffrement des fichiers, ainsi que l'adresse où se trouvent ces fichiers.

L'application offre une interface qui s'affiche sur un navigateur Web installé sur la machine locale à laquelle la carte à puce est connectée (*figure 3.2*). Via cette interface, un ensemble de services est offert. Par exemple l'utilisateur peut accéder à la liste des noms de fichiers stockés dans des serveurs distants, ajouter ou supprimer un fichier. Elle offre aussi la possibilité d'accéder à la liste des serveurs où sont stockés ces fichiers, d'ajouter ou de supprimer des serveurs.

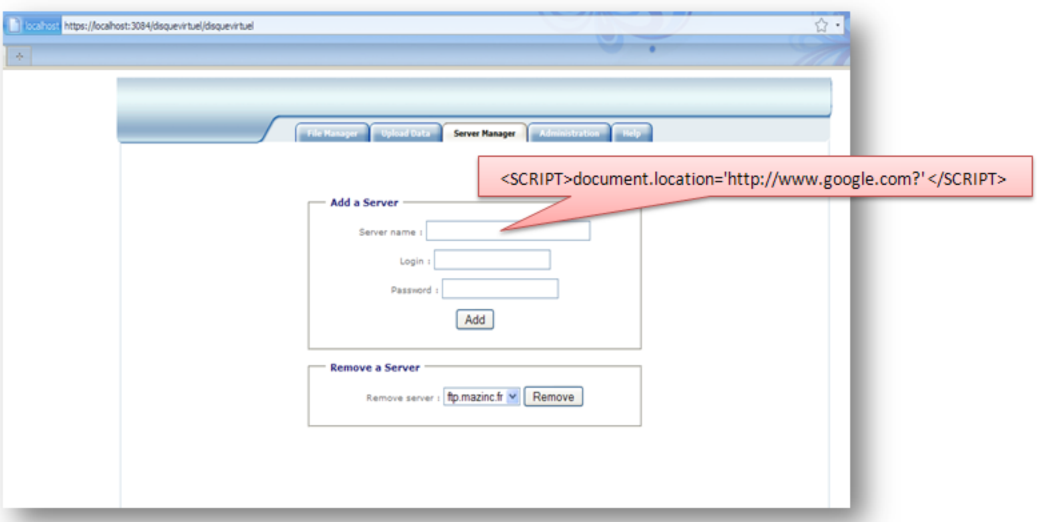

Fig. 3.2 – Attaque XSS persistante

En supposant que cette application ne prévoit aucun filtrage des entrées, nous avons réussi à causer un déni de service sur l'application. Nous avons utilisé une attaque XSS persistante qui a rendu l'application pratiquement inutilisable. Dans le champ destiné à ajouter un nom de serveur, nous avons inséré du code JavaScript dont l'interprétation redirige l'utilisateur vers un autre site google.com par exemple. Le script est le suivant :

```
<SCRIPT>document.location='http://www.google.com?'+document.cookie
</SCRIPT>
```

La liste des serveurs étant stockée en mémoire persistante et utilisée dynamiquement par un ensemble de pages Web de notre application, il suffit que l'utilisateur tente de se rendre sur une de ces pages pour que le script soit exécuté et que l'utilisateur soit redirigé vers google.com. Ainsi, son application devient inutilisable. Pour la réutiliser correctement, il est nécessaire de recharger une nouvelle instance de l'application dans la carte.

Nous pouvons imaginer d'autres scénarios plus complexes qui permettraient de récupérer des informations sensibles.

3.2 Portée et exploitation des attaques XSS

Les dernières applications Web offrent de larges possibilités d'accès aux différents éléments d'une page Web, notamment grâce à l'utilisation du JavaScript. Souvent l'attaquant tente d'exploiter ces différents points d'entrée afin d'avoir des privilèges (ou d'élever ses privilèges) ou de prendre le contrôle du navigateur Web d'une victime. L'injection de codes malicieux dans une application Web peut avoir des conséquences plus ou moins graves, telle qu'une simple défiguration du site, une redirection de l'utilisateur vers un site sous le contrôle de l'attaquant, le vol d'informations sensibles ou l'accès à des privilèges.

Plusieurs sites Web, dont certains sont très populaires, ont été victimes d'attaques XSS. Souvent les victimes évitent de dévoiler qu'elles ont subi des attaques, notamment quand il s'agit de site Web commerciaux. En 2005, des pirates ont exploité une vulnérabilité dans *gmail.com* pour usurper l'identité d'abonnés légitimes et compromettre leur domaine. En 2006, le site *CBS News* a été victime de la publication d'une fausse information. En 2008, Sarah Palin, en pleine campagne présidentielle aux États Unis, a été victime d'une attaque XSS sur sa boite e-mail Yahoo, et des informations sélectionnées à partir de son compte ont été postées sur *Wikileaks*. Le site de la compagne présidentielle 2008 de Obama a aussi été victime d'une redirection vers le site Web de son adversaire Hillary Clinton [XSSa].

Un grand classique des XSS, qui a affecté plusieurs sites Web de grande notoriété comme *paypal.com* [XSSc] et quelques sites relevant du domaine defense.gouv.fr [XSSb], consiste à rediriger les utilisateurs vers un faux site (une copie de l'original) qui est sous le contrôle de l'attaquant. Souvent l'utilisateur ne fait pas attention à l'URL de la page Web qu'il consulte, il risque alors d'entrer des informations confidentielles (coordonnées bancaires par exemple) dans ce faux site, qui vont être ensuite envoyées à l'attaquant.

Des réseaux sociaux comme *FaceBook, MySpace*, etc. ont également révélé des vulnérabilités dans le passé. Ces vulnérabilités sont liées à une faiblesse ou à une absence de filtrage des données. Dans *FaceBook* par exemple la vulnérabilité se situait au niveau des commentaires associées aux photos.

XSSF [Lud11] est un logiciel d'attaques XSS. Il permet de détecter des failles XSS et de démontrer leur dangerosité. XSSF commence par créer un lien avec les victimes afin d'envoyer des attaques par la suite. Un ensemble d'attaques (alerte, vol de cookie, keylogger [7], tabnapping [8],

7. dispositif qui charge les frappes de touches du clavier et les enregistre, à l'insu de l'utilisateur
8. modifier, à l'insu de l'utilisateur et sans aucune intervention de sa part, le contenu affiché dans un onglet

etc.) est stocké sous forme d'un module MSF (fichier structuré contenant le code à exécuter). Ces travaux ont démontré à travers de simples injections, les possibilités énormes des attaques XSS, qui peuvent aller jusqu'à l'exploitation à distance des failles du système d'exploitation et la prise de contrôle (totale) d'une machine distante. Cette prise de contrôle est possible dès lors que le navigateur ou l'un de ses plugins est vulnérable à une faille de sécurité.

Le système d'exploitation Android (utilisé sur les smartphones) a également été victime d'une vulnérabilité XSS. Thomas Cannon [Tho10] a détecté une vulnérabilité qui permet de récupérer depuis un site Web malicieux (ou vulnérable aux XSS) n'importe quelle information stockée sur la carte SD du téléphone. Dans [MSP11] est présentée une attaque XSS permettant l'installation non assistée des applications arbitraires de Android Market.

Souvent les attaques XSS, notamment les volatiles, nécessitent d'utiliser des méthodes permettant d'inciter l'utilisateur à cliquer sur un lien hostile. Parmi ces méthodes, les techniques d'ingénierie sociale (l'envoi d'un e-mail semblant provenir d'un contact sûr, ou faire croire à une situation d'urgence, etc), le clickjacking [9], qui consiste à utiliser des liens invisibles ou cachés sous des liens légitimes, ou le DNS rebinding [10] susceptible de permettre de transformer le navigateur de la victime en un simple relai entre le réseau local et une machine contrôlée par l'attaquant. Afin de masquer la présence d'un code qui pourrait faire douter l'utilisateur, les pirates utilisent des techniques d'encodage des URL [Gar09]. Par exemple, plutôt que d'inviter la victime à visiter la page suivante où nous pouvons apercevoir le code JavaScript :

```
index.php?query=enter+your+search+terms+here&type=advanced&results=
10&searchType=3&action=search&page=33"><script>alert(document.cookie)
</script>
```

un attaquant va utiliser un encodage comme suit :

```
index.php?query=enter+your+search+terms+here&type=advanced&results=
10&searchType=3&action=search&page=%33%33%5c%22%3e%3c%73%63%72%69%70%
74%3e%61%6c%65%72%74%28%64%6f%63%75%6d%65%e%74%2egetElementById(\OT1\
textquoterightmyScript%63%6f%6f%6b%69%65%29%3c%2f%73%63%72%69%70%74%3e
```

3.3 Solutions existantes

Les vulnérabilités XSS sont connues depuis plusieurs années, mais elles sont sous estimées par beaucoup de développeurs et d'utilisateurs qui pensent qu'elles permettent simplement d'afficher une boite de dialogue, alors qu'il ne s'agit que d'une preuve de concept. Nous avons vu dans la section précédente le potentiel des attaques XSS et que beaucoup de sites Web en ont été victimes d'attaques dues à une vulnérabilité XSS. Pendant que les attaques se multiplient sur le Web, de nombreuses recherches tentent de trouver des solutions pour détecter ou prévenir ce type de vulnérabilité. Parmi les solutions proposées, nous avons celles basées du côté client (navigateur Web), et d'autres du côté serveur (serveur où l'application Web est installée).

9. technique cherchant à inciter un utilisateur à cliquer sur un lien malicieux présenté comme sûre.
10. manipulation des informations DNS afin de prendre à distance le contrôle d'un routeur Internet.

3.3.1 Solutions côté client

Un filtrage XSS effectué entièrement du côté client permet à des navigateurs Web de vérifier des entrées utilisateur au préalable, avant leur envoi au serveur, ou de vérifier les réponses du serveur avant qu'elles soient ne interprétées par le client. Les solutions côté client peuvent consister en modules additionnels intégrés aux navigateurs Web ou des mécanismes externes de détection d'intrusion installés sur le même équipement que le navigateur Web, jouant le rôle d'un proxy qui contrôle le trafic au niveau applicatif (requêtes/réponse HTTP) entre l'application Web et le navigateur Web.

1. **Renforcement des navigateurs Web**

   L'approche la plus simple et la plus utilisée pour se prémunir des attaques XSS consiste à renforcer le navigateur Web. Plusieurs navigateurs Web (IE8, Mozila Firefox, Google Chrome, etc) sont dotés d'une couche additionnelle de protection ou proposent des modules additionnels pour se prémunir de ces attaques.

   Microsoft a ajouté en 2008 un module anti-XSS dans le navigateur IE 8 ; il consiste à scanner les paramètres d'une requête qui peuvent être malveillants. Quand de tels paramètres sont sélectionnés, IE8 génère dynamiquement une expression régulière (une éventuelle réponse du serveur). Il contrôle ensuite la réponse du serveur. Si une correspondance complète entre la réponse du serveur et une expression régulière est vérifiée alors une attaque XSS réfléchie est en cours. Le filtre modifiera donc automatiquement cette réponse de telle sorte à faire échouer l'attaque. Cependant, si l'attaque n'est pas correctement neutralisée, un script malveillant peut être exécuté. Ce filtrage peut non seulement être facilement outrepassé, mais il peut également rendre l'application vulnérable. En effet, des recherches [ED] ont démontré que l'outil pouvait rendre des applications vulnérables à XSS, en manipulant les réponses modifiées par IE8 pour effectuer de simples abus.

   Un autre type de protection conçu pour Mozila Firefox est un module complémentaire : NoScript [MOZ]. Ce module est basé sur une liste blanche. Il permet de limiter l'exécution de scripts JavaScript à des domaines de confiance sélectionnés par l'utilisateur. Le système effectue un blocage préventif des scripts dont le domaine n'appartient pas à la liste blanche. Cependant, cette solution est très restrictive. Les sites Web de nos jours utilisent beaucoup de code JavaScript. NoScript oblige donc des interventions répétées de l'utilisateur qui doit à chaque fois choisir de bloquer ou d'autoriser les scripts d'un domaine. De plus, des scripts malicieux dans un site de confiance ne peuvent pas être détectés [Won].

   Une autre approche utilisée dans Netscape et présentée par Vogt et al [FNE+07] consiste à utiliser la technique d'étiquetage des données en entrée du navigateur Web. L'objectif est de veiller à ce que les données sensibles (cookies, identifiants de session), relatives à un domaine ne soient pas transférées à un domaine tiece. Par exemple, quand une page Web contient un formulaire HTML, les valeurs entrées par l'utilisateur sont étiquetées indiquant leur serveur d'origine. L'étiquette est propagée à toute donnée générée à partir de la manipulation d'une donnée étiquetée. Ainsi, si un script malveillant s'exécutant dans une autre fenêtre tente d'espionner ce formulaire HTML et de faire des copies des entrées de l'utilisateur, les valeurs copiées vont également être étiquetées et ne pourront pas être envoyées à un autre serveur que celui d'origine. Comme pour le module NoScript de Mozila Firefox, c'est à l'utilisateur

de choisir d'autoriser ou de bloquer un transfert de données.

2. **Systèmes de détection d'intrusion côté client**

   Des solutions basées sur un mécanisme de détection d'intrusion appelées pare-feu ou proxy applicatifs se placent entre le client et le serveur. Ils permettent d'analyser les requêtes et réponses échangées entre les deux parties en vérifiant que les réponses ne contiennent pas de scripts non-autorisés ou que des informations sensibles ne sont pas envoyées à un domaine malicieux.

   Dans [ECGN06], une méthode de filtrage côté client est proposée pour empêcher que le navigateur Web de la victime contacte des URLs malveillantes. Leur outil appelé *Noxes* est un proxy qui utilise une "liste noire" contenant un ensemble de liens qui ne sont pas de confiance. Ainsi, le proxy contrôle les requêtes et les réponses, et bloque toute redirection ou envoi de données sensibles vers des liens appartenant à cette "liste noire". Nous considérons que cette méthode n'est pas suffisante pour détecter ni empêcher les attaques XSS complexes. Seules les XSS volatiles fondées sur la violation de la politique de « même origine » [Fou06] pourraient être détectées. Des techniques alternatives d'attaques XSS, comme celle proposée dans [Lud11], peuvent être utilisées afin de contourner un tel mécanisme de prévention. De plus Noxes nécessite une configuration par son utilisateur qui doit spécifier les règles de filtrage (liens malicieux).

   Les auteurs dans [OMYS04], présentent un autre proxy côté client qui contrôle et analyse des données échangées entre le navigateur et le serveur. Leur processus d'analyse consiste à vérifier les caractères composant les paramètres des requêtes et réponses envoyées et émises par le serveur, et à détecter les caractères malicieux qui pourraient être interprétés par le navigateur Web. Pour cela, le proxy vérifie si des caractères malicieux présents dans une requête se retrouvent dans la réponse du serveur. Si c'est le cas, deux modes de fonctionnement sont possibles. Dans le premier, la réponse est encodée et transmise au client avec un message d'alerte. Dans le deuxième, la requête est encodée puis envoyée à nouveau au serveur. La réponse du serveur est ensuite retransmise au client avec un message d'alerte. De toute évidence, la principale limitation d'une telle approche est qu'elle ne peut être utilisée que pour empêcher les attaques XSS non persistantes. De plus, elle engendre un temps de réponse supplémentaire.

3.3.2 Solutions côté serveur

Les solutions côté serveur, peuvent être divisées en deux approches : la programmation préventive qui intervient à la phase de développement de l'application, et les mécanismes de détection d'intrusion (proxy ou pare-feu applicatif) installés sur le serveur et qui contrôlent les flux échangés entre le serveur et le client. Il existe également d'autres outils permettant de vérifier la bonne implémentation des applications Web, appliquant une approche basée sur la programmation préventive.

1. **Programmation préventive**
   La programmation sécurisée (programmation préventive) est une forme de conception défensive qui vise à assurer la fonction permanente de logiciels en dépit des utilisations imprévisibles. L'approche de programmation préventive fournit un soutien sous forme d'API

(*Application Programming Interfaces*) ou de bibliothèques qui peuvent être utilisées par les développeurs. Deux API bien connues pour la protection contre les attaques XSS sont *htmLawed* [htm] et *Kses* [Kse]. Elles sont destinées à des applications écrites en langage PHP.

Kses est un script de filtrage HTML/XHTML écrit en PHP. Ce filtre est utilisé par l'application de blogs *WordPress*. Le but de cette API est de supprimer tous les éléments et attributs HTML indésirables afin d'atténuer les attaques XSS, *buffer overflows*, etc. *htmLawed* est un autre filtre, fournit en logiciel libre, qui nettoie du code HTML pour éviter les failles de sécurité. Il réécrit essentiellement le code HTML pour supprimer les balises et les données qui pourraient être utilisées pour effectuer des attaques XSS. Cependant, de nombreuses attaques XSS ne sont pas empêchées, comme l'injection de code HTML par le biais des expressions de feuilles de style CSS. De plus, l'approche qui consiste à supprimer les caractères malicieux peut être facilement contournée. Si l'API supprime le mot <script> dans un vecteur d'entrée, la chaîne suivante : <scr<script>ipt>alert(document.cookie)</scr</script>ipt> génère une XSS qui ne va pas être détectée.

D'autre part, une collection de logiciels et de méthodes de classes et qui pourraient également inclure différentes API, forment généralement une bibliothèque. Deux des bibliothèques anti-XSS les plus populaires sont *AntiSamy OWASP* [Ant] et *Microsoft Anti-Cross Site Scripting Library* [Lam06]. *AntiSamy* est une bibliothèque logicielle libre, qui analyse et nettoie les entrées HTML/CSS en utilisant une technique de validation basée sur une liste blanche. Elle utilise un fichier XML (liste blanche) qui définit les balises et les attributs autorisés dans une application. *AntiSamy* a de très bonnes capacités de filtrage XSS et son API est propre. Cependant, elle présente un inconvénient dû à la difficulté de créer et de maintenir le fichier XML pour chaque application. Un autre inconvénient d'*AntiSamy* est qu'elle ne prend pas en charge la totalité des balises HTML/CSS, telles que les balise d'images.

La bibliothèque Microsoft Anti-XSS [Mica] fait partie de la bibliothèque *Microsoft Web Protection (WPL)*. Elle se base aussi sur l'approche de liste blanche pour encoder les entrées utilisateur non fiable (HTML, l'attribut HTML, XML, CSS et JavaScript). L'avantage de cette bibliothèque par rapport à son équivalente open-source, AntiSamy, est qu'elle est relativement plus facile à utiliser, et peut être appliquée à toute application ASP.NET sans exiger beaucoup de modifications du code existant. Toutefois, cette bibliothèque ne peut être utilisée que pour des projets codés avec la technologie .NET. Au contraire, l'API *AntiSamy* a été porteé sur différentes plateformes telles que .NET et PHP.

L'organisation OWASP [11] propose une autre bibliothèque appelée *Entreprise Security API* (ESAPI). ESAPI aide les développeurs à se protéger des défauts de sécurité lors de la conception et de la mise en œuvre, en fournissant un ensemble commun d'interfaces pour les contrôles de sécurité dont : l'authentification, le contrôle d'accès, la validation des entrées, l'encodage des sorties, etc.

Ces interfaces sont conçues de manière à être simple à comprendre et à utiliser et prennent en charge, automatiquement, de nombreux aspects de sécurité d'applications. La bibliothèque est disponible pour différents langages de programmation, ce qui permet d'avoir une interface unique entre les différents langages. Un développeur en Java ou .NET qui a déjà utilisé ESAPI passera rapidement à la version PHP.

Quel que soit l'approche de sécurité de codage, les techniques qui devraient être utilisées

11. https ://www.owasp.org/index.php/Main_Page

pour atténuer les attaques XSS sont la validation (filtrage) et l'encodage (échappement) des entrées et sorties. Ces techniques et leurs variantes sont décrites ci-dessous :

– **Validation des entrées / sorties**
  La validation des entrées est une technique utilisée pour vérifier si l'entrée utilisateur est un contenu non-valide ou dangereux. Dans une telle approche, les données entrées par un utilisateur sont scannées et comparées soit à une liste blanche soit à une liste noire. Une action appropriée est ensuite appliquée. L'utilisation d'une « liste blanche » garantit que toutes les requêtes sont refusées, sauf si elles sont spécifiquement autorisées. D'autre part, la « liste noire » contient l'ensemble des entrées connues comme malveillantes. Cette approche est moins robuste que la précédente car les modèles d'attaque XSS sont en constante évolution.
  Cependant, comme mesure de protection supplémentaire, il peut être intéressant d'utiliser les deux techniques à la fois. Les filtres de liste noire sont insuffisants, mais cela ne signifie pas qu'ils sont inutiles. Une approche hybride lorsqu'elle est mise en œuvre correctement peut entraîner une défense très efficace contre les attaques XSS. La liste blanche sert à s'assurer qu'une entrée correspond à un format désigné, et la liste noire permet d'exclure d'autres problèmes connus.
  La validation des sorties fonctionne de la même manière que la validation des entrées, mais seules les données destinées à être affichées dans une page Web sont scannées. En règle générale, la validation des entrées/sorties est appliquée dans le but d'atténuer le risque d'attaques XSS, et l'approche de la liste noire est la plus utilisée.

– **Encodage des entrées / sorties**
  L'encodage est une autre technique de filtrage de contenu. Elle consiste essentiellement à rendre sans danger des caractères, des tags et des scripts indésirables, en les remplaçant par une représentation alternative [20007], encodée (par exemple <script> encodé en HTML : <script>). Ce type de filtrage peut être utile lorsque l'espace de saisie autorisé ne peut pas garantir que l'entrée est sûre. L'encodage permet de traiter des entrées potentiellement malicieuses comme du texte plutôt qu'un script exécutable.
  Tout comme la validation, il y a deux variantes d'encodage : l'encodage des entrées et l'encodage des sorties.
  L'encodage des entrées oblige l'utilisateur à connaître le contexte dans lequel ces données vont être utilisées. Ceci est adapté pour de petites applications Web, mais il peut être compromettant pour les applications Web mises en œuvre par plusieurs développeurs, surtout si l'application est distribuée. La prédiction du contexte d'utilisation des données pourrait impliquer des hypothèses qui pourraient introduire de nouvelles vulnérabilités. D'autre part, l'encodage des sorties a tendance en général à être plus facile à mettre en œuvre. Le développeur a besoin de déterminer le contexte dans lequel les données sont retournées. En règle générale, seul l'encodage des sorties est utilisé comme un moyen de défense XSS.
  L'encodage peut parfois être détourné par les pirates pour créer des vecteurs XSS brouillés qui contournent la plupart des filtres de validation basés sur une liste noire. Le tableau 8.1, présente les différentes façon d'écrire la chaîne <SCRIPT>alert("XSS")</SCRIPT> en utilisant des codages différents. Les chaînes qui en résultent démontrent la difficulté, voir l'impossibilité, d'avoir une liste noire qui contiendrait toutes les combinaisons possibles

d'entrées invalides. Cependant, l'interprétation de ces différents encodages dépend aussi du navigateur Web utilisé [OWAc].

Encodage	**Chaine résultante**
HTML	`<script>alert('xss')</script>`

Hex	`<script`
	`>alert('`
	`xss')</s`
	`cript>`

Decimal	``ĕ™ĔąĒ` `Ėb—ĈāĔĖ` `@9Ġĕĕ9A` ``Gĕ™ĔąĒ` `Ėb`

Octal	`tţŃŢőŠŤ`
	`vŁŔŅŢŤP`
	`GŰţţGQt`
	`WţŃŢőŠŤ`
	`v`

UTF-7	`+ADw-script+AD4-alert('XSS')+ADsAPA-/SCRIPT+AD4`
Base64	`PHNjcmlwdD5hbGVydCgnWFNTJyk7PC9TQ1JJUFQ+Cg==`

Table 3.1 – Exemples d'encodages pour outrepasser une liste noire

Il est en général convenu, que le codage sécurisé est l'action la plus naturelle et la plus souhaitable pour prévenir et corriger les vulnérabilités XSS. Cependant, les développeurs ont besoin d'être formés ou habitués à écrire du code sécurisé. Ils sont généralement tenus de développer des applications fournissant un maximum de services et qu'elles soient conviviales et ergonomiques, le tout dans un délai restreint, ce qui ne leur laisse pas le temps de se soucier de la sécurité. L'utilisation d'une bibliothèque tierce pourrait être d'une grande aide à ces développeurs. En outre, l'encodage sécurisé engendre l'inconvénient de l'efficacité réduite due au temps supplémentaire lié à la validation et/ou l'encodage. L'application utilisant à la fois la validation et l'encodage des entrées/sorties peut conduire à une dégradation des performances au niveau du serveur. Les spécialistes en sécurité considèrent que pour des raisons de performance, le filtrage des entrées peut être facultatif mais les sorties doivent être nécessairement filtrées.

2. **Systèmes de détection d'intrusion et pare-feu applicatifs**

   Tout comme les solutions côté client, il existe des proxys et des pare-feux applicatifs destinés à être installés sur le serveur qui héberge l'application à protéger. *ModSecurity* [Mod] est un pare-feu applicatif *open-source* qui agit comme un module pour le serveur Web Apache. Il intercepte les requêtes envoyées et les réponses reçues par le serveur web, les traite et les filtre à base d'une « liste noire » contenant des règles de requêtes non souhaitées. Selon les auteurs, *ModSecurity* n'a pas d'impact sur les performances même quand il est lancé en parallèle avec d'autres périphériques réseau. Il offre une bonne protection contre certaines des attaques les plus courantes en utilisant sa liste noire définie par défaut. Cependant, l'inconvénient majeur de *ModSecuriy* est qu'il ne supporte pas les différents algorithmes de codage, les filtres peuvent donc être contournés.

   Scott et Sharp [SS02] ont développé un langage de description de politique de sécurité (*SDLP : Security Policy Description Langage*) qui spécifie des contraintes de validation et des règles

de transformation définissant ce qu'il faut faire (réaction) quand un certain motif (pattern) est détecté dans une entrée utilisateur. Ce système met en place un pare-feu applicatif qui utilise une base de données de signatures, relatives à des entrées malicieuses. Bien que ce système donne l'assurance immédiate de la sécurité des applications Web, il présente deux inconvénients principaux. Le premier est la difficulté d'identifier et de créer des contraintes de validation pour chaque point d'entrée spécifique à une application Web. Le deuxième est l'augmentation du temps de réponse du serveur qui ralentit la génération dynamique de pages Web.

PHPIDS [PHP] est un IDS *open-source*. Cette IDS permet de protéger les applications PHP des injections XSS, SQL, et d'autres attaques. PHPIDS est simple à utiliser et offre une couche de sécurité bien structurée pour les applications Web PHP. Contrairement à *ModSecurity*, cet outil prend en charge divers algorithmes de codage. En effet, PHPIDS peut détecter des attaques brouillées, utilisant différents ensembles de caractères comme UTF-7, JavaScript Unicode, les entités décimales et hexadécimales. Toutefois, il présente les inconvénients de : générer une quantité importante de faux positifs, d'être dépendant du langage PHP et de consommer beaucoup de cycles CPU.

Bien que ces outils fournissent une assurance dynamique et donc immédiate de la sécurité des applications Web, dans la plupart des cas et dans certaines mesures, la plupart des solutions basées sur les IDS et les pare-feux applicatifs peuvent être contournés avec de simples astuces comme indiqué dans le tableau 8.1. Ce tableau montre quelques-unes des attaques XSS qui, selon Eduardo Alberto Vela Nava et al. [Nav10] ne sont pas détectées par les IDS. La raison est que la plupart de ces outils se basent principalement sur un modèle de sécurité négatif : « la liste noire ». Or, il existe différentes façons d'écrire une attaque XSS qui ne peuvent pas toutes être définies dans une liste noire. Cette approche protège les applications Web uniquement à la phase de déploiement au lieu d'essayer de contribuer à éliminer les failles pendant la phase de développement.

3. **Stratégies d'évaluation de la sécurité**
   L'évaluation de la sécurité est un point de vue rigoureux dans le développement d'applications. Elle consiste à rechercher des vulnérabilités potentielles qui peuvent permettre des attaques sur l'application.

   Généralement, l'analyse statique et/ou l'analyse dynamique sont les méthodologies utilisées pour effectuer l'évaluation. Deux des outils les plus représentatifs qui utilisent cette approche pour les XSS sont WebSSARI [HYH+04] et Pixy [NCE06].

   L'analyse statique présente l'inconvénient de générer un taux élevé de faux positifs, qui engendre souvent une analyse manuelle par la suite. L'analyse dynamique présente l'inconvénient de générer des faux négatifs et d'être couteuse, nécessitant généralement l'instrumentation du code analysé. De plus, les outils d'analyse et d'évaluation de logiciels sont liés au langage de développement. La plupart de ces outils s'intéressent à des applications web écrites en PHP et il n'existe aucun outil d'analyse d'applications Web pour carte à puce.

   Dans le chapitre suivant, nous présentons cette approche plus en détails.

3.4 Conclusion

Dans ce chapitre, nous avons étudié un cas spécifique d'attaques sur les applications Web. Nous avons mis l'accent sur les risques des vulnérabilités XSS et les différentes approches pour s'en prémunir. L'approche basée au niveau client présente l'avantage d'apporter une solution plus généraliste, en protégeant l'accès à un ensemble d'applications consultées à partir du navigateur Web qui l'implémente. Cependant, la protection offerte est peu fiable et peut être facilement contournée. D'autre part, les solutions côté serveur ne nécessitent pas d'intervention de l'utilisateur, et ont beaucoup moins de risque d'être contournées que les solutions côté client.

Nous considérons que la meilleure prévention contre des attaques par injection de code, particulièrement les XSS, consiste à sensibiliser les développeurs à appliquer une méthodologie de développement sécurisée en intégrant une vérification et un encodage des entrées et sorties de l'application.

Les solutions de vérification et d'évaluation des applications, permettent de certifier que l'application est correctement développée, appliquant les règles de sécurité définies par une méthodologie. Nous présentons un état de l'art de ces solutions dans le chapitre suivant.

Chapitre 4

Détection de vulnérabilités dans un programme

Sommaire

4.1 Introduction **53**
4.2 Analyse statique **55**
4.2.1 État de l'art des outils d'analyse statique 56
4.2.2 L'analyse statique dans Java Card 58
4.3 La dépendance causale **59**
4.3.1 Description 59
4.3.2 Travaux existants 60
4.4 Conclusion **61**

4.1 Introduction

Les conséquences importantes liées aux vulnérabilités des programmes ont incité à multiplier les efforts de recherche pour le développement d'outils de détection d'erreurs de programmation et de failles de sécurité. Il existe différentes techniques de vérification et de test de programmes, contribuant ainsi à l'amélioration de leur qualité. Nous distinguons trois approches : des outils basés sur un assistant de preuve, le test et l'analyse de programmes.

Les outils basés sur un assistant de preuve offrent à l'utilisateur un moyen de construire des preuves et vérifient ensuite, de manière plus ou moins automatique qu'elles sont correctes dans le programme associé. ESC/Java est un outil basé sur cette approche qui a été largement utilisé dans l'analyse des applications Java. Il a été développé par *Compaq SRC*. Il permet de trouver des erreurs de programmation Java. L'utilisateur doit définir une spécification des classes et méthodes de l'application qu'il souhaite analyser. ESC/Java analyse et vérifie si l'implémentation satisfait cette spécification. Un prouveur de théorèmes [DJ04] est utilisé pour prouver si la spécification cible est une conséquence logique de la spécification source. Si la spécification n'est pas respectée, l'outil lance une alerte. La spécification est écrite sous forme d'annotations Java et représentée

par des invariants de classe et des pré- et post-conditions d'une méthode. La performance de ce type d'outils est fortement liée à la qualité de la spécification formelle qui est spécifique à une application et doit être définie par l'utilisateur.

La validation de logiciels à base de tests consiste à appliquer un jeu de tests sur un programme et de vérifier qu'il fonctionne correctement. Le *Fuzzing* est une technique de test très utilisée dans l'industrie. Elle consiste à injecter un ensemble de tests invalides, inattendus ou aléatoires, en entrée d'une application dans le but de mettre à défaut son exécution (générer : une exception, un crash, un résultat inattendu). Le *Fuzzing* peut s'appliquer en « boite blanche » ayant le code source, en « boite grise » se basant sur le code binaire ou « boite noire » sans avoir aucune connaissance de l'application testée. Il existe différents logiciels de *Fuzzing* pour la détection de vulnérabilités par injection de code dont WAPITI [WAP] et WEBFUZZER [WEB]. Cependant, cette technique ne peut pas être complète dans le sens où il est impossible d'avoir tous les cas de tests possibles, notamment dans des cas comme les attaques XSS où un même caractère peut être représenté de différentes façons, en utilisant différents encodages.

L'analyse d'un programme est un processus de vérification automatique de son comportement dans le but de détecter s'il est correct : respectant les règles et la syntaxe du langage de développement utilisé, optimisé ou encore sécurisé en appliquant des méthodologies de développement bien définies. On distingue deux classes d'analyses : l'analyse statique et l'analyse dynamique.

1. **L'analyse statique :** permet la détection et l'élimination d'erreurs de programmation dès les phases préliminaires de conception, et par conséquent d'affiner le programme testé ou d'améliorer sa robustesse. Cette technique présente l'avantage d'être indépendante de la plate-forme d'exécution, elle est donc universellement applicable. Cependant, il a été mathématiquement démontré à base de résultats fondés sur le théorème de Rice [12] que l'analyse statique d'un programme est un problème indécidable. En effet, dans certains cas il est difficile de désigner la suite d'exécutions possibles du programme. Dès qu'il contient des boucles ou des branchements conditionnels (while, if, etc.), il devient difficile de décrire le chemin d'exécution ou de savoir si le programme se termine. Cependant, l'application d'un ensemble de principes (contrôle des dépassements de tampon par exemple) lors de la phase de conception, permet de réduire le risque d'erreurs à l'exécution. En outre, l'analyse statique a l'avantage de faciliter la maintenance du code.
2. **L'analyse dynamique :** consiste à analyser l'exécution d'un programme. Elle est souvent basée sur l'instrumentation des applications par l'ajout d'instructions permettant le suivi et le contrôle des flux d'informations [FNE+07, CW09]. L'instrumentation peut être réalisée au niveau du code source, lors de la compilation, ou à partir du code binaire ou du bytecode ; elle permet de suivre les flux d'informations internes des applications.

Les mécanismes de détection d'intrusion peuvent être vus comme des outils d'analyse dynamiques. Leur objectif est de surveiller les flux émanant et reçus par une application pendant son exécution, et de détecter des cas de violation de politiques de sécurités spécifiées. Les travaux proposés par Guillaume Hiet nous ont suscité un grand intérêt. L'outil d'analyse dynamique JBlare [Gui08, GVLB08] qu'il propose effectue une analyse dynamique des flux au niveaux applicatif et système. Cet outil utilise un moniteur externe à l'application permettant le suivi de la propagation des flux entre les différentes applications et entre les applications et le système. Le moniteur est

12. le théorème de Rice dit que toute propriété non triviale (c'est-à-dire qui n'est pas toujours vraie ou toujours fausse) sur la sémantique dénotationnelle d'un langage de programmation Turing-complet est indécidable.

installé au niveau de l'environnement d'exécution des programmes (Machine Virtuelle Java JVM, système d'exploitation,. . .) ce qui nécessite souvent des modifications ; une instrumentation des applications s'impose aussi.

Synthèse : l'approche dynamique présente l'avantage d'être plus précise que l'approche statique en générant moins de « faux positifs » (fausses alertes), mais elle ne garantit pas la couverture totale du code. En effet, la validation de ce type d'outils est basée sur une suite de tests, or il est souvent difficile de prévoir tous les cas possibles de tests, ce qui engendre des «faux négatifs » (cas non traités).

D'autre part, l'instrumentation du bytecode et la modification de la JVM présentent un coût considérable en termes d'espace mémoire et de temps d'exécution supplémentaires générés par l'analyse. Les travaux présentés dans JBLARE [Gui08, GVLB08] ont démontré la grande efficacité de cette technique permettant de détecter des violations de politiques de sécurité par suivi des flux de contrôle au niveau applicatif et au niveau du système d'exploitation. Toutefois, les résultats expérimentaux dans un environnement PC montrent que cette analyse augmente le temps d'exécution de plus 5%, ce qui ne convient pas dans un environnement à faibles ressources tel qu'une carte à puce.

Pour ces différentes raisons et afin de vérifier la robustesse de l'implémentation des applications Web Java Card 3, nous avons choisi de concevoir un outil d'analyse statique. Cette approche est plus compatible pour un système à capacités restreintes (la carte à puce par exemple). De plus, elle permet de réduire les risques d'erreurs et de vulnérabilités, et de détecter des failles de sécurité avant de charger les applications dans le système.

4.2 Analyse statique

L'analyse statique est un moyen de vérification automatique permettant de prédire les comportements d'un programme lors de son exécution sans réellement l'exécuter. Elle est généralement utilisée dans le débogage, en particulier pour la recherche d'erreurs pouvant apparaître à l'exécution. Il s'agit de la technique la plus adéquate pour analyser des applications dédiées à des plateformes à ressources réduites telles que les cartes à puce où une analyse dynamique serait très coûteuse.

L'analyse statique consiste généralement à explorer les différents chemins d'exécution possibles d'un programme. Elle se base sur la construction d'un graphe de flux de contrôle (*Control flow graph*, CFG) qui est une représentation par un graphe de tous les chemins qui peuvent être traversés dans un programme durant son exécution. Les nœuds du graphe représentent les blocs d'instructions de base qui sont toujours exécutées sans interruption et les arêtes représentent les flux de contrôle qui relient deux blocs différents. Un graphe d'appels représente potentiellement un flux de contrôle inter-méthodes. Les nœuds d'un graphe d'appels correspondent aux méthodes et une arête représente une méthode qui invoque une autre méthode.

Une analyse plus élaborée inclut également un contrôle des flux de données échangés entre des méthodes ou des applications. Elle permet d'avoir des informations sur l'ensemble des données pouvant apparaitre en un certain point du programme. Elle est généralement utilisée afin de détecter

des objets non référencés dans le programme ou de vérifier que certains objets sensibles ne sont pas utilisés dans un contexte non-prévu ou non-sécurisé.

Il existe différentes techniques d'analyse statique, la plus répandue consistant à définir des motifs correspondant à une vulnérabilité ou des erreurs liées à un langage de programmation et de rechercher ces motifs dans le code. D'autres analyses plus élaborées sont basées sur l'analyse des flux de données et des flux de contrôle. Enfin, il existe des outils qui nécessitent d'ajouter des annotations dans le code pour spécifier une règle à vérifier.

4.2.1 État de l'art des outils d'analyse statique

Plusieurs recherches se sont intéressées à la vérification de programmes Java. Certaines se situent dans la catégorie des outils qui fournissent des informations syntaxiques concernant des règles stylistiques d'écriture de programmes [Che]. D'autres avertissent de l'usage de constructions considérées comme dangereuses ou interdites par des règles visant à une programmation plus «sûre». Nous nous intéressons à cette deuxième catégorie d'outils.

Wagner et al. [SJCP05] présentent une comparaison entre un ensemble de ces outils, dont *Findbugs* [HP04], *PMD* [PMD], *QJ PRO* [QJP] et quelques projets industriels (non cités). Leur étude se base sur la quantification des « faux positifs » générés par chaque outil. Dans [RAF04], le nombre de « faux négatifs» est aussi pris en compte. Ces deux comparaisons ont conclu que chaque outil est en mesure de détecter des bogues que les autres ne détectent pas, et par conséquent la combinaison de ces différents résultats permettrait d'avoir une analyse plus complète. Cependant, certains de ces outils comme *Findbugs* sont évolutifs. Ils utilisent des modules complémentaires, chacun étant en charge de vérifier une propriété. Dans nos travaux nous avons sélectionné l'outil d'analyse *Findbugs*, nos critères de sélection étant basés sur :

- l'analyse de flux de données : afin de suivre la propagation des données douteuses dans le code ;
- l'extensibilité : permettant d'ajouter facilement des modules pour adapter l'outil à notre besoin de détecter des attaques Web XSS ;
- le logiciel libre (Open source) qui permet d'apporter des ajouts et modifications.

Dans la suite nous présentons quelques exemples de logiciels d'analyse statique, *open source*, notamment l'outil *Findbugs* :

JLint. JLint [Kon] est un outil d'analyse de bytecode Java qui effectue des contrôles syntaxiques, une analyse de flux de données et une analyse inter-procédurale des programmes à multi-tâches. Il contient un composant appelé AntiC, qui est un vérificateur de syntaxe pour les langages de la famille du langage C, à savoir : C/C++, objectiveC et le langage Java. Cependant, JLint ne permet pas l'analyser des servlets ce qui est contraignant dans notre cas d'étude, puisque nous nous intéressons à la sécurité des applications Web Java Card 3 (basées sur des servlets). De plus, cet outil n'offre pas de mécanisme d'intégration de plugins (modules additionnels). Pour ajouter une nouvelle propriété d'analyse, il est nécessaire de modifier le code source et de développer les méthodes de test supplémentaires en langage C.

PMD. PMD [PMD] permet d'analyser du code source Java. Il est basé sur la définition d'un ensemble de règles à rechercher dans un code. Un ensemble de règles est fourni par défaut dans l'outil, à savoir :

- absence d'un des états try / catch / finally ;
- les variables locales, les paramètres et les méthodes privées non utilisées ;
- les objets qui ne sont pas nécessaires ;
- les expressions trop compliquées - inutiles si les déclarations, les boucles for qui pourraient être des boucles while ;
- la duplication de code.

L'outil est extensible, permettant de développer de nouvelles règles à inclure dans l'analyse. Toutefois le développement de ces règles nécessite des connaissances en Arbre syntaxique abstrait (*Abstract Syntax Tree*, AST) sur lequel PMD se base.

***Findbugs* :** *Findbugs* [Fin] est un logiciel, distribué dans les termes d'une licence LGPL et développé à l'université du Maryland, Etats-Unis. Il permet de détecter des bogues dans un programme Java, à partir de l'analyse du bytecode à la recherche de certains motifs.

Findbugs inclut un ensemble important de détecteurs permettant de déceler des bogues généralement liés à des défauts de programmation. Cependant, l'outil est extensible offrant la possibilité de rajouter des plugins pour vérifier des propriétés supplémentaires. L'utilisateur peut limiter son analyse à uniquement des propriétés qu'il souhaite vérifier en sélectionnant les détecteurs qui l'intéressent.

Une synthèse effectuée dans [LL05] compare un ensemble d'outils d'analyse statique et classe l'analyseur *Finbugs* comme l'outil le plus facilement extensible qui offre la possibilité d'ajouter un plugin pour vérifier une propriété de sécurité additionnelle. Dans [IBM], un développeur d'IBM décrit en détail comment développer un nouveau plugin.

Un plugin est une archive au format « jar » qui peut comporter un ou plusieurs détecteurs. Les détecteurs sont basés sur le patron `Visitor`. Ils implémentent l'interface `Detector` qui inclut la méthode `visitclassContext()` qui invoque chaque classe de l'application. Le logiciel utilise la librairie BCEL (*Byte Code Engineering Library*) qui est une librairie Java permettant d'analyser et de manipuler des fichiers « class » Java et d'obtenir des informations élémentaires sur tous les objets et méthodes contenus dans ces fichiers (méthodes, attributs, instructions, etc).

Findbugs propose un ensemble de classes et d'interfaces qui peuvent être étendues pour le développement d'un nouveau plugin. Dans notre cas, nous avons exploité la classe `OpcodeStack`, qui permet de calculer une abstraction de la pile d'exécution en fonction des différentes instructions bytecode (Opcode) possibles. L'analyse des flux de données dans *Findbugs* tient compte de la dépendance causale entre les variables locales d'une méthode (analyse intra-procédurale). Cependant, cette analyse de dépendance causale se limite à une analyse méthode par méthode et ne tient pas compte des appels interméthodes d'une même classe ou de classes différentes. D'autre part, l'analyse intra-procédurale est incomplète car différents cas d'*Opcode* ne sont pas complètement traités, notamment dans le cas des tableaux et des vecteurs (`aaload, aastore`, ...).

4.2.2 L'analyse statique dans Java Card

Les applets Java Card sont des applications destinées à être utilisées dans des domaines assez sensibles comme dans les cartes bancaires. De ce fait, différentes recherches se sont intéressées à la vérification de ce type d'applications. Compte tenu des ressources limitées des plateformes (cartes à puces, système embarqués, ...) qui hébergent ces applets, la plupart des recherches se sont basées sur l'analyse et la vérification des applications avant d'être chargées dans la carte (*off-Card*). Parmi ces outils, de nombreux travaux comme Jack [BRlLV03], Loop [JB01], ou encore les travaux de Gemplus présentés dans [CnH02], se basent sur la vérification formelle et utilisent le langage JML (*Java Modeling Langage*) pour la définition d'une spécification de l'ensemble des classes et interfaces de l'application. JML (*Java Modeling Language*) est un langage de spécification d'un comportement. Il permet de définir l'usage correct de l'ensemble des applications (interface syntaxique du code, signature des méthodes, types des attributs, etc.) se basant sur la définition des pré et post-conditions ainsi que des invariants. Les spécifications sont ajoutées dans une application Java ou Java Card sous forme d'annotations Java. Cependant, ce type d'outils demande un effort considérable de la part du développeur qui doit lui-même définir la spécification correspondante à l'application qu'il souhaite analyser.

Dans [PJM+00, JL00], les auteurs présentent le projet PACap, dont l'objectif est de vérifier les interactions entre des applets Java Card, et de certifier qu'une nouvelle applet interagit de manière sécurisée avec des applets préalablement chargées dans la carte. L'outil proposé vérifie les flux de données entre des objets de la carte à puce par une analyse statique. Il contrôle si une application est correctement implémentée respectant une politique de sécurité qui définit les niveaux de partage autorisés. Cependant, l'outil ne peut pas détecter d'autres violations que celles définies dans la politique de sécurité.

Loizidis et al. [ALK+08] présentent un outil d'analyse statique automatique basé sur plusieurs détecteurs qui étendent l'analyseur *Findbugs*, pour vérifier trois types de violations. Le premier détecteur est permet de verifier le respect des restrictions relatives aux communications inter-applets via une interface de partage. Il construit un CFG pour toutes les classes d'une applet afin de détecter les violations de propriétés inter-procédurales, telles que des appels récursifs à une méthode déclarée dans les interfaces de partage. Le second détecteur vérifie que les appels à des méthodes d'API ne provoquent pas d'exception non gérée, qui pourrait conduire à quitter l'applet dans un état imprévisible. Le troisième permet d'effectuer une analyse de flux de données pour vérifier les arguments d'une méthode invoquée et détecter des failles tels que des paramètres nuls ou non valides (une valeur négative pour un index dans un tableau).

Notre étude se base sur *Findbugs*, associé à la technique de dépendance causale « tainting ». L'objectif de notre analyse est de tracer la propagation des données dans une exécution abstraite de bytecode et de vérifier la présence de vulnérabilités à des attaques Web par injection de code dans des applications Web dédiées à des cartes à puce, en particulier les Java Card 3. Nous verrons par la suite qu'un ensemble d'améliorations dans *Findbugs* a été nécessaire afin de rendre notre analyse plus efficace.

4.3 La dépendance causale

L'impact de la non-validation des données en entrée d'un système ou d'une application peut être inestimable. Les attaques par injection de données malicieuses sont très répandues mais elles deviennent difficiles à réaliser quand le développeur prévoit un filtrage et une validation des données avant de les utiliser dans des fonctions sensibles. L'analyse de dépendance causale est une technique qui permet de vérifier que ces fonctions sont présentes dans un code.

4.3.1 Description

La dépendance causale est la technique qui permet de suivre la propagation de données potentiellement malicieuses afin de vérifier qu'elles ne sont pas utilisées dans des contextes sensibles à la sécurité. Une dépendance causale entre deux objets est présente, s'il existe un flux d'informations direct ou indirect entre eux. Cette technique consiste d'abord à marquer par une étiquette les données en entrée du logiciel et ensuite de propager cette étiquette à d'autres objets calculés en fonction d'objets étiquetés. Une alerte est lancée quand l'étiquette est propagée à des objets (ou paramètres) inappropriés où la sécurité pourrait être compromise.

Livshits et al. [LL05] définissent la dépendance causale par un ensemble d'états : source, dérivation, puits, et neutralisation décrits comme suit :

Source : correspond aux points d'entrée d'un programme par lesquels un utilisateur peut introduire des données. Dans les différents langages de programmation, il existe un ensemble de méthodes permettant de retourner les données entrées par l'utilisateur (exemple : la méthode *getParametter()* dans Java). Dans les applications Web nous avons des méthodes permettant de retourner des données HTML, de lire les cookies enregistrés au niveau du client ou d'analyser des paramètres HTML. La dépendance causale commence par associer une étiquette à toute donnée issue d'un état source.

Dérivation : les données issues d'un état source peuvent potentiellement être manipulées pour générer d'autres données qui peuvent être elles-mêmes manipulées pour générer d'autres données. La manipulation peut consister en diverses opérations : une assignation, une concaténation, une soustraction, etc. Toute donnée dérivée directement ou indirectement d'une donnée source appartient à un état dérivation et tout objet d'un état dérivation doit être étiquetée.

Puits : certaines méthodes critiques nécessitent que les arguments qu'elles manipulent soient « sûres », telle que la méthode `connect()` qui permet de se connecter à une base de données. Si une telle méthode manipule une donnée étiquetée générée directement ou indirectement (via un état dérivation) à partir d'un état source, alors il existe une potentielle vulnérabilité. L'état correspondant à cette situation est appelé *puits*.

Neutralisation : Des langages comme PHP prévoient des fonctions de validation permettant de vérifier ou de valider des données issues d'un état source ou dérivation. Ces fonctions pourraient être également développées par le développeur de l'application ou par une tierce personne. L'état

correspondant à l'appel à une telle méthode est appelé *neutralisation*. Une étiquette associée aux données générées par un état *neutralisation* doit être retirée.

4.3.2 Travaux existants

La plupart des travaux basés sur la dépendance causale est dynamique, la propagation des étiquettes se faisant pendant l'exécution du code (programme). Cette technique est introduite dans Perl [Wal00]. Pour l'appliquer, il faut utiliser le mode *Taint*. Les données extérieures (les entrées utilisateur, les entrées variables d'environnement et les fichiers en entrée du programme) sont explicitement marquées par une étiquette (*taint*). Ces données ne doivent pas être utilisées directement ou indirectement dans une commande qui invoque le *shell* ou une commande qui crée ou modifie un fichier, un répertoire ou un processus. Perl offre aussi un mécanisme qui permet d'annuler le marquage des données préalablement étiquetées en définissant des expressions régulières correspondant à des données valides. L'utilisateur doit lui-même définir ces expressions régulières, Perl rend ensuite valide tout objet étiqueté (annulation de l'étiquette) qui correspond à des expressions régulières. Cependant, l'objectif du mode *Taint* de Perl n'est pas de détecter des attaques mais des erreurs de programmation non-intentionnelles.

Ruby [TCF05] propose une analyse à granularité plus fine que Perl. La dépendance causale est appliquée au niveau des objets et non simplement au niveau des variables de type chaîne de caractères. Tout objet dont un attribut est étiqueté au cours de l'exécution du programme est également étiqueté. Toutefois dans ce cas l'annulation de l'étiquette devient plus complexe. En effet l'annulation de l'étiquette d'un objet revient à vérifier que tous les attributs le constituant ne sont pas marqués par une étiquette.

La dépendance causale a également été utilisée dans beaucoup de travaux d'analyse d'applications Web, pour la détection de vulnérabilités à des attaques Web par injection de code telles que XSS et SQL injection. Ces outils sont généralement appliqués à des applications Web écrites en langage PHP [NCE06, HYH+04]. Halfond et al. [WAP08] proposent un suivi au niveau des caractères. Leur système nécessite une instrumentation (réécriture) du *bytecode*. Dans [CW09], Chin et al s'inspirent de ces mêmes travaux mais contrairement à l'approche présentée dans [WAP08], les modifications sont portées au niveau des bibliothèques Java au lieu de la réécriture du *bytecode* de l'application. L'outil proposé est donc plus facile à déployer et plus compatible par rapport au code existant. Sur le plan conceptuel, pour chaque chaîne String, une étiquette est associée à chaque caractère la constituant, indiquant s'il est dérivé de données vérifiées ou pas. Les classes de la bibliothèque Java relatives aux objets `String (String, StringBuffer, et StringBuilder)` sont instrumentées afin d'enregistrer cette information d'étiquette et de pouvoir la propager. Toutefois l'analyse au niveau caractère peut être facile à contourner. En effet, des attaques par injection de code sont souvent très subtiles : un caractère écrit dans un encodage particulier pourrait être combiné avec un autre caractère et générer une donnée malicieuse.

Livshits et al [LL05, MVBL05] proposent un outil d'analyse hybride (statique et dynamique) destiné à des applications JEE *(Java Entreprise Edition)*. Les vulnérabilités sont définies dans un langage de requêtes appelé PQL*(Program Query Language)* dont la syntaxe est proche du langage Java. Une requête PQL est une suite d'événements incluant l'approche de propagation des étiquettes en fonction des différents événements. Une vulnérabilité est détectée en cas de présence d'une correspondance entre le programme analysé et une requête. Dans l'analyse statique le programme

est traduit en bddbddb [13] [JL04] et les requêtes sont converties en langage Datalog [14]. L'analyse dynamique permet de vérifier les résultats de l'analyse statique et de réduire le taux de « faux positifs ». Elle est basée sur l'instrumentation de bytecode. Afin de réduire les coûts de cette instrumentation, seuls les cas détectés par l'analyse statique sont pris en compte.

Les requêtes doivent être écrites par l'utilisateur pour représenter la vulnérabilité recherchée. Dans le cas d'une correspondance entre le programme et une requête, deux choix sont offerts, soit de se limiter à lancer une alerte, soit de remplacer les méthodes critiques non sécurisées, par une autre plus sécurisée. Le code source de cet outil est disponible. Nous l'avons testé sur l'application Java fournie en exemple et sur une application Web basique. Comme dans [SAP06], nous avons constaté que l'outil d'analyse statique n'est pas complet : aucun outil de conversion des requêtes en Datalog n'est fourni. D'autre part, l'analyse dynamique donne des résultats sur les applications Java mais elle n'est pas fonctionnelle pour des applications Web basées sur des servlets.

Nous avons été particulièrement intéressés par l'approche de Livshits et al [LL05, MVBL05] ; notamment l'analyse dynamique qui permet de remplacer une méthode non sécurisée par une autre méthode plus sécurisée. Nous pourrions par exemple remplacer une méthode
`HTTPServletResponse.getWritter().println()`
par une méthode qu'on appellerait `printAfterValidation`() qui filtrerait les données avant de les envoyer au navigateur Web. Cependant, nous avons constaté que les coûts de cette analyse sont beaucoup trop importants pour être supportés par la carte à puce. Les coûts supplémentaires de cette approche incluent l'instrumentation du bytecode, la taille de l'outil d'analyse qui doit être chargé dans la carte et aussi les fonctions sécurisées qui vont être utilisées pour remplacer des fonctions qui ne le sont pas. Vu ces différentes contraintes nous avons dû abandonner cette solution pour nous consacrer à une analyse statique basée sur l'outil *Findbugs*.

Findbugs propose une analyse de flux de données intra-procédurale en calculant une abstraction de l'exécution du bytecode. L'outil fait une simulation du processus d'exécution méthode par méthode, sans tenir compte des différents appels inter-méthodes ou interclasses. L'analyse n'est donc ni inter-procédurale ni interclasses. Au niveau d'une méthode, l'analyse inclut des paramètres permettant d'appliquer l'approche de dépendance causale. Nous avons utilisé cet outil que nous avons amélioré pour faire une analyse plus complète, interclasses et inter-procédurale, pour des applications Web Java Card 3. Nous présentons notre contribution dans le chapitre suivant.

4.4 Conclusion

La sécurité des applications Web contre des attaques par injection de code doit être prise en compte depuis le développement. Les données en entrée doivent être considérées comme potentiellement malicieuses et l'application doit inclure des filtrages de données pour vérifier, filtrer, et valider ces données avant qu'elles ne soient utilisées par des fonctions critiques. L'analyse statique de bytecode permet de vérifier la bonne implémentation de ces applications et cela, avant de les charger dans la carte à puce.

Associée à la technique de dépendance causale, l'analyse statique de bytecode permet de vérifier si les données douteuses sont filtrées avant d'être utilisées en paramètres de fonctions critique ou

13. Approche basée sur les diagramme de décision binaire dans la représentation des programmes Java
14. Datalog est un langage de requête et de règles pour les bases de données déductives

pas. Cependant ce type d'analyse est connu pour être exposé à des « faux positifs » liés à des cas indécidables où l'abstraction de l'exécution du code peut ne pas être précise. A l'opposé, l'analyse dynamique est plus sûre mais nécessite souvent une instrumentation du bytecode. Toutefois, cette instrumentation présente deux inconvénients. En plus du risque de générer un disfonctionnement de l'application en modifiant le contexte d'exécution du code original, l'instrumentation génère des coûts d'espace mémoire et de temps d'exécution très important, sans compter le coût relatif à l'analyse proprement dite. Les conséquences de ces coûts sont particulièrement plus considérables dans un environnement contraint comme les cartes à puce. Pour ces différentes contraintes, nous avons préféré opter pour une analyse statique que nous présentons dans le chapitre suivant.

Chapitre 5

Analyse statique de bytecode Java Card 3

Sommaire

5.1 Présentation générale 64
5.2 Représentation d'une vulnérabilité XSS 64
5.3 Processus d'exécution d'un *bytecode* 65
5.3.1 Zones de données 66
5.3.2 La frame 67
5.3.3 Les instructions de la JCVM 67
5.4 *Findbugs* et ses limites 69
5.5 Analyse inter-procédurale et interclasses 70
5.5.1 Présentation du concept de CFG 70
5.5.2 Analyse inter-méthodes et inter-classes 72
5.5.3 Imprécision de l'analyse insensible au contexte 72
5.6 Analyse des flux de données 73
5.6.1 La dépendance causale 73
5.6.2 Étiquetage des conteneurs 75
5.6.3 Détection des champs persistants 78
5.6.4 Dépendance causale inter-méthodes 78
5.7 API de filtrage XSS pour des applications cartes à puces 79
5.7.1 Filtrage des données insérées dans un élément HTML 80
5.7.2 Filtrage des données insérées dans des attributs HTML 80
5.7.3 Filtrage des données insérées dans un code *JavaScript* 81
5.7.4 Filtrage et validation des données insérées dans les propriétés de style HTML 82
5.7.5 Filtrage des données insérées dans des valeurs de paramètres d'URL . . 83
5.8 Conclusion 83

5.1 Présentation générale

Dans ce chapitre nous présentons la première partie de nos travaux (voir chapitre 8 pour la deuxième partie qui aborde le *fuzzing* sur le protocole HTTP). Notre approche se base sur une analyse statique du *bytecode* pour détecter des vulnérabilités par injection de code dans des applications Web dédiées à des cartes à puce. Nous nous intéressons particulièrement aux applications Java Card 3 [KLIC11]. L'approche proposée est basée sur l'outil d'analyse *Findbugs*. L'analyse effectuée par *Findbugs* présente plusieurs limites et un grand taux de faux positifs et de faux négatifs. Nous avons amélioré cette analyse par la construction d'un graphe d'appel complet, incluant les invocations de méthodes de même ou de différentes classes, tout en tenant compte du contexte d'exécution de chaque méthode. Afin de suivre la propagation des données dans une application, nous avons effectué une analyse intra et inter procédurale de flux de données.

Notre analyse se base sur une représentation abstraite du fonctionnement de la JCVM (Java Card Virtual Machine) et une interprétation abstraite du processus d'exécution d'un programme. Nous utilisons la technique de dépendance causale dans le suivi des flux de données.

Nous avons développé un module (*plugin*) *XSSDetector* additionnel au logiciel *Findbugs* qui permet de détecter des failles dues au non filtrage des données non fiables au niveau de tout point d'insertion ou elles pourraient être interprétées. Une faille XSS est représentée par un automate à états finis. Une vulnérabilité XSS est détectée si une correspondance complète existe entre un chemin d'exécution du programme et l'automate.

Pour se prémunir des attaques XSS le développeur doit prévoir des fonctions de filtrage. Afin de réduire la tâche aux développeurs et d'éviter les risques de failles dues à un mauvais filtrage, nous mettons à leur disposition une API *JCXSSFilter*. Cette API comporte un ensemble de fonctions permettant de neutraliser des données potentiellement malicieuses. Le développement de *JCXSSFilter*, respecte les recommandations de l'organisation OWASP.

Le module *XSSDetector* vérifie que l'API *JCXSSFilter* est utilisée dans tous les points d'insertion d'un programme où un filtrage est nécessaire.

5.2 Représentation d'une vulnérabilité XSS

Les vulnérabilités XSS peuvent être représentées sous forme d'un automate à états finis (figure 5.1) composé principalement de quatre états :

- I : état Initial ;
- S : état Source ;
- D : état Dérivation ;
- F : état Final.

L'invocation de méthodes retournant des données externes au programme fait passer l'automate de l'état I à un état S. Parmi ces méthodes, nous trouvons : `HttpServletRequest.getParametter()`, `HttpServletRequest.getAttribute()`, etc. La manipulation des données d'un état S par des méthodes autres que des fonctions de filtrage, telles que : `_.append(x), new String(x), new StringBuffer(x), x.toString(), x.substring(_ ,_), x.toString(_)`, etc (x étant une donnée générée d'un état S ou D) fait passer l'automate à l'état D.

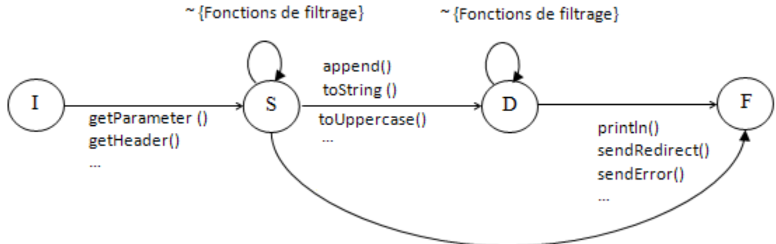

Fig. 5.1 – Automate d'une vulnérabilité XSS

La condition de passage à l'état final est différente selon le type de la vulnérabilité XSS :

- Dans le cas d'une XSS persistante : l'état final est atteint si les données générées dans un état D sont stockées dans un champ persistant ;
- Dans le cas d'une XSS volatile : l'état final est atteint, en cas d'invocation de méthodes permettant d'afficher des données générées par un état D dans une page Web telles que : `HttpServletResponse.println()`, `HttpServletResponse.sendError()`, `HttpServletResponse.setHeader()`, etc.

Atteindre un état final signifie qu'une vulnérabilité XSS est présente. Dans les deux cas, volatile ou persistante, une vulnérabilité XSS est détectée s'il existe dans le programme analysé au moins une correspondance avec l'automate XSS.

5.3 Processus d'exécution d'un *bytecode*

Notre approche étant basée sur une analyse statique, elle effectue une analyse du *bytecode* sans l'exécuter réellement. Elle se base sur la simulation de toutes les structures de données que génère la JCVM, dans l'exécution d'un programme.

Une JCVM est une machine abstraite définie par une spécification. Elle permet d'exécuter des *bytecode* Java Card. Lors de son lancement, la JCVM initialise les espaces de données nécessaires à l'exécution du programme. Le tas (*Heap*) et l'espace des méthodes sont les principaux espaces. Chaque thread du *bytecode* possède une pile privée où sont sauvegardées les frames. Une frame est créée à chaque invocation d'une méthode. Elle consiste en une pile d'opérandes, un tableau de variables locales et une référence au pool de constantes de la classe de la méthode courante.

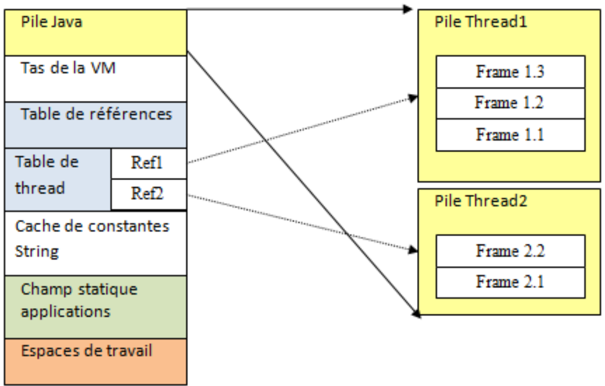

Fig. 5.2 – Représentation des structures de la JCVM

5.3.1 Zones de données

La mémoire de la machine virtuelle est organisée en zones de données dont : le tas, la zone de méthodes et les piles Java (pile de *threads*) permettant de stocker différents types de données tels que les objets créés, les variables locales de méthodes, les opérandes des méthodes, les valeurs de retour de méthodes, etc. Il existe aussi des piles de méthodes natives pour les codes natifs appelés par les applications Java.

Certaines zones sont créées au démarrage de la JCVM et sont détruites seulement lorsque celle-ci est arrêté. D'autres zones de données sont utilisées pour chaque *thread*, et ont la même durée de vie que le *thread* associé. Dans ce qui suit, nous présentons brièvement ces différents espaces de stockage de données.

1. ***Le registre PC (program counter)***

   La machine virtuelle peut exécuter plusieurs *threads* à la fois. Chaque *thread* possède son propre *PC*. Quand le *thread* exécute une méthode, *PC* indique l'adresse de la prochaine instruction du *bytecode* à exécuter.

2. ***La pile de la JCVM***

   Chaque *thread* de la JCVM dispose d'une pile privée, créée en même temps que le *thread*. La pile est composée d'un ensemble de frames [Apa06]. À chaque invocation de méthode, une nouvelle frame est créée et empilée au sommet de la pile Java du *thread* associé. Elle est dépilée et détruite à la fin de l'exécution de la méthode.

3. ***Le tas***

   C'est une zone mémoire commune à tous les *threads* de la machine virtuelle, où sont stockés tous les objets instances de classes ou de tableaux. A chaque objet d'instance est associée une référence qui permet d'accéder aux valeurs de chaque champ et aux méthodes associées à la classe de l'objet. La libération d'espace dans le tas est gérée automatiquement par le ramasse-miettes.

4. ***Zone des méthodes***

   La JCVM possède un espace de méthodes qui est partagé par tous les *threads*. Cet espace contient le *bytecode* des méthodes, les constructeurs et des informations telles que la table des symboles de constantes, tables de méthodes, champs de classe. Elle contient aussi les informations de typage sur les objets (nom, modificateur, nom de la superclasse, nombre et noms des interfaces implémentées, nombre de méthodes) et pour chaque méthode, un descripteur (nom, modificateur, nombre et type de paramètres, type du résultat, table des exceptions, etc.)

5. ***La table des constantes***

   La table des constantes est allouée dans la zone des méthodes, pour chaque classe ou interface créée par la machine virtuelle Java. Elle contient plusieurs sortes de constantes telles que les valeurs de chaînes de caractères, les valeurs de constantes entières, noms symboliques de classes, interfaces, méthodes, les variables de classe ou d'instance, etc.

6. ***La pile des méthodes natives***

   Les méthodes natives sont des méthodes écrites dans un autre langage que Java (C/C++, assembleur) et compilées vers un langage natif de la machine sous-jacente. Ces méthodes

sont chargées dynamiquement lors d'une invocation par un programme Java. Notre analyse ne s'étend pas à ce type de méthodes. Nous nous limitons aux méthodes définies dans l'application analysée.

5.3.2 La frame

Une frame est générée à chaque invocation d'une méthode. Elle permet de stocker des données nécessaires à l'exécution de la méthode et/ou des résultats partiels de méthodes (des valeurs de retour ou des exceptions). Elle est composée d'un ensemble de structures de données à savoir :

- **La table de variables locales** : stocke les paramètres et les variables locales de la méthode. Les variables locales sont adressées par index. L'index de la première variable locale est 0. Les valeurs de type `long` ou `double` occupent deux variables locales consécutives. Lors de l'invocation d'une méthode d'instance, la variable locale d'index 0 est utilisée pour référencer l'objet sur lequel la méthode a été invoquée (*this* en langage Java) et les variables locales sont indexées de manière consécutive à partir de l'index 1.
- **La pile des opérandes** : contient les résultats partiels des calculs intermédiaires ou des valeurs de retour d'invocation d'autres méthodes.
- **Des informations complémentaires** : il peut s'agir d'un registre contenant l'adresse de la prochaine instruction à exécuter, appelé communément *program counter* (pc), d'une référence vers la table des constantes de la classe à laquelle appartient la méthode, d'une référence vers la table d'exceptions de la méthode, etc.

5.3.3 Les instructions de la JCVM

Notre analyse étant basée sur une interprétation abstraite de l'exécution du *bytecode*, il est nécessaire de connaitre les différentes instructions d'un *bytecode* et comment ces dernières sont interprétées dans la JCVM.

Une instruction de la machine virtuelle Java est composée d'un *opcode* sur un octet, spécifiant l'opération à effectuer, suivi par zéro ou plusieurs opérandes servant d'arguments qui seront utilisés par l'opération. Pour une grande partie des *opcodes*, le mnémonique qui leur est associé est préfixé par une lettre représentant le type de la donnée associée à ces instructions. Par exemple, la lettre "i" correspond à la manipulation d'une donnée de type « integer » (entier signé de taille 32 bits signé). Dans ce qui suit, nous présentons quelques exemples d'instructions.

Instructions de chargement et de stockage

Les instructions de chargement et de stockage permettent de transférer les valeurs entre les variables locales et la pile d'opérandes d'une frame de la JCVM. Ces instructions peuvent être classées comme suit :

1. *Instructions de chargement de variables locales dans la pile des opérandes :* iload, iload_N, lload, lload_N, fload, fload_N, dload, dload_N, aload, aload_N.
2. *Instructions de chargement d'une valeur depuis la pile des opérandes vers une variable locale :* istore, istore_N, lstore, lstore_N, dstore, dstore_N, astore, astore_N.

3. *Instructions de chargement de constantes dans la pile des opérandes :* bipush, sipush, ldc, ldc_w, ldc2_w, aconst_null, iconst_m1, iconst_N, lconst_N, dconst_N.
4. *Les instructions qui accèdent aux champs des objets :* (putstatic, getstatic, putfield, getfield) et aux éléments des tableaux (baload, caload, saload, iaload, laload, daload, aaload, bastore, castore, sastore, iastore lastore, dastore, aastore), ces instructions transfèrent aussi des données depuis et vers la pile des opérandes.

Le format des instructions avec des lettres génériques N désigne des familles d'instructions spécifiques à une instruction générique qui ne prend qu'un paramètre.

Instructions de manipulation de la pile d'exécution

Ces instructions permettent de manipuler directement le sommet de la pile : pop, pop2, dup, dup2, dup_x1, dup2_x1, dup_x2, dup2_x2, swap.

Les instructions arithmétiques

Les instructions arithmétiques prennent deux valeurs sur la pile des opérandes et chargent le résultat au-dessus de cette même pile. Parmi ces instructions :

- Addition : `iadd, ladd, dadd`.
- Soustraction : `isub, lsub, dsub`.
- Multiplication : `imul, lmul, dmul`.
- Division : `idiv, ldiv, ddiv`.
- Reste (modulo) : `irem, lrem, drem`.
- OR (bit à bit) : `ior, lor`.
- AND (bit à bit) : `iand, land`.
- XOR (bit à bit) : `ixor, lxor`.
- Incrémentation des variables locales : `iinc`.
- Comparaison : `dcmpg, dcmpl, lcmp`.

Création d'objet

La machine virtuelle crée les instances de classe et les tableaux de façon distincte avec un jeu d'instructions propre à chacun :

- Création d'une nouvelle instance de classe : `new`.
- Création d'un nouveau tableau : `newarray, anewarray, multianewarray`.

Les instructions de branchement

Les instructions de branchements conditionnels ou inconditionnels permettent à la JCVM de continuer à s'exécuter avec une instruction différente de celle suivant l'instruction de branchement. Les instructions de branchement sont :

- Branchement conditionnel : `ifeq,iflt, ifle, ifne,ifgt, ifge, ifnull, ifnonnull, if_icmpeq, if_icmpne,if_icmplt,if_icmpgt, if_icmple, if_icmpge, if_acmpeq, if_acmpne`.
- Branchement conditionnel composé : `tableswitch, lookupswitch`.
- Branchement inconditionnel : `goto, goto_w, jsr, jsr_w, ret`.

Instructions d'invocation de méthode

Les instructions suivantes permettent d'invoquer une méthode :

- **Invokevirtual** : invoque une méthode d'un objet, en appelant la bonne méthode virtuelle de l'objet.
- **Invokeinterface** : invoque une méthode qui est implémentée par une interface, en cherchant les méthodes implémentées par cet objet pour trouver la méthode appropriée.
- **Invokespecial** : invoque une instance qui requiert un traitement spécial ; une méthode d'initialisation d'instance, une méthode privée, ou une méthode de la super classe.
- **Invokestatic** : invoque une méthode de classe statique.

Les instructions de retour de méthodes

Elles se distinguent par leur type de données de retour. L'instruction `return` est utilisée pour un retour depuis des méthodes déclarées void, des méthodes d'initialisation d'instance, et des méthodes d'initialisation de classe ou d'interface. les instructions `dreturn, areturn` et `ireturn` sont utilisées pour retourner des valeurs de type `boolean, byte, char, short, ou int`.

Lancement d'exception

Une exception est lancée par le programme en utilisant l'instruction `athrown` ou par d'autres instructions de la JCVM lorsqu'une condition inhabituelle est détectée (par exemple, la division par zéro).

5.4 *Findbugs* et ses limites

Notre analyse est basée sur l'outil *Findbugs* qui offre une représentation abstraite de l'ensemble des structures composant une JCVM. L'algorithme ci-dessous présente le fonctionnement de *Findbugs* dans l'analyse d'un *bytecode*. Toutes les classes de l'application sont visitées séparément. Le détecteur va ensuite itérer pour chaque méthode définie dans la classe (même si cette dernière n'est jamais invoquée dans le programme) et génère une frame lui correspondant pour effectuer une analyse des flux de données. Si la méthode courante invoque une autre méthode, les traitements de la méthode invoquée sont ignorés. La frame de la méthode courante est détruite à la fin de l'analyse.

Algorithm 1 Analyse statique par Findbugs

```
1: Parcourir les classes de l'application
2: for Chaque classe do
3:    lire l'ensemble des méthodes définies dans la classe courante
4:    for chaque méthode do
5:       Détruire la frame précédemment construite si elle existe
6:       générer la frame de la méthode courante
7:       for  chaque instruction de la méthode do
8:          effectuer une analyse des flux de données
9:          mettre à jour la frame suivant le type de l'instruction visitée
10:         if instruction== invocation d'une autre méthode then
11:            ajouter un nouveau item dans la pile des opérandes
12:            pas de traitement de la méthode invoquée à ce niveau
13:         end if
14:      end for
15:   end for
16: end for
```

L'interprétation abstraite du processus d'exécution dans *Findbugs* est limitée. Le suivi des flux de contrôle n'est pas interclasses et l'analyse des flux de données se limite au niveau d'une méthode et ne tient pas compte des invocations imbriquées entre méthodes. Par conséquent, ce logiciel dans son état ne conviendrait pas à nos besoins. Il serait impossible de suivre la propagation des données dans tout le programme. Notamment dans le cas de données passées en paramètre d'une méthode ou de variables de retour. Les champs de classe ne serait pas bien gérées aussi (champs définis par le modificateur `static`). Les champs de classes devraient être modifiables par toutes les méthodes de la classe et une modification devraient être vue par l'ensemble de ces méthodes. D'autre part, nous avons constaté que certaines instructions ne sont pas complètement gérées par *Findbugs*, notamment les instructions de gestion des conteneurs telles `aaload` et `astore`.

Nous avons amélioré *Findbugs* par une analyse des flux de contrôle et des flux de données qui est à la fois intra-procédurale, inter-procédurale, inter-classes et également sensible au contexte. Notre analyse de flux de données applique la technique de dépendance causale pour le suivi de la propagation des données dans un programme. Nous avons développé un module (*plugin*) de détection des vulnérabilités XSS persistantes et volatiles.

5.5 Analyse inter-procédurale et interclasses

Afin de réaliser une analyse inter-procédurale qui inclut les appels entre des méthodes de même ou de différentes classes, nous avons construit un graphe de flux de contrôle (CFG), sensible au contexte et global à toute l'application.

5.5.1 Présentation du concept de CFG

Le CFG (*Control flow graph*) est une représentation sous forme de graphe de tous les flux qui constituent les différents chemins d'exécution possibles d'un programme. Un flux de contrôle désigne l'ordre dans lequel sont exécutées les instructions ou les fonctions d'un programme. Il peut être modifié ou interrompu par divers types d'instructions qui peuvent engendrer la séparation des

flux en un ou plusieurs chemins. Dans Java ou particulièrement Java Card, ces instructions peuvent être divisées en plusieurs catégories :

- Branchement inconditionnel : l'exécution se poursuit à une instruction différente.
- Branchement conditionnel : dans ce cas, il existe au maximum, une alternative d'exécution. Une condition doit être satisfaite afin qu'un ensemble d'instructions puisse être exécuté.
- Branchement conditionnel à choix multiples : ce sont des branchements conditionnels avec plus d'une alternative d'exécution.
- Boucle : consiste à exécuter zéro ou plusieurs fois un bloc d'instructions tant qu'une condition n'est pas encore satisfaite.
- Arrêt inconditionnel : constitué des instructions capables d'arrêter l'exécution d'un programme tels que les retours de fonction ou les levées d'exceptions.

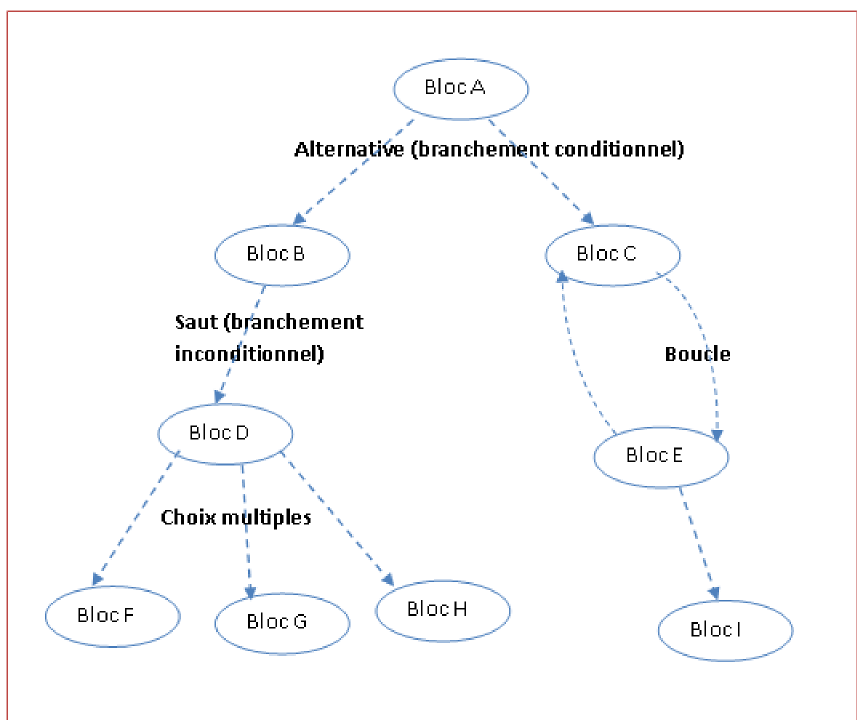

Fig. 5.3 – Exemple de graphe CFG

Le graphe de flux de contrôle est constitué de *nœuds* et *d'arêtes*. Les nœuds correspondent aux blocs élémentaires qui sont constitués d'une succession d'instructions ne contenant aucune rupture du flux de contrôle et aucune cible d'instruction de branchement. Ce bloc possède un point d'entrée et un point de sortie. Ainsi si un bloc est exécuté, toutes les instructions le composant seront exécutées sans interruption. Le point d'entrée du bloc peut être atteint par plusieurs autres blocs, les liens entre ces différents blocs sont représentés par des arrêtes qui relient les nœuds correspondants.

5.5.2 Analyse inter-méthodes et inter-classes

Afin de suivre la propagation des données dans une application, il est nécessaire de connaître tous les chemins d'exécution possibles incluant les appels entre des méthodes de même classe ou de classes différentes. Afin de pallier aux limites de *Findbugs* nous avons construit un graphe d'appel inter-classes sensible au contexte.

L'algorithme *Algorithm 2* représente le processus de génération du graphe d'appel interclasses. La construction commence par lister l'ensemble des méthodes définies dans l'ensemble des classes constituant l'application. Nous sauvegardons une description qui distingue chaque méthode par son nom, son type, le nom de classe où elle est définie et le contexte de l'objet auquel elle appartient. Un graphe de flux de contrôle est construit pour cette méthode. Le code de chacune des méthodes enregistrées est parcouru séparément. Si la méthode courante invoque une autre méthode, une arrête est construite entre les deux nœuds correspondant à ces méthodes. Ainsi, en itérant l'opération pour chaque méthode de l'application, nous obtenons un graphe complet incluant tous les appels interclasses.

Algorithm 2 Graphe d'appels inter-classes

Lister l'ensemble des méthodes de toutes les classes
Construire un nœud pour chaque méthode appartenant à un contexte de classe
for Chaque classe du programme **do**
  for chaque méthode de la classe **do**
    rechercher toute invocation d'une autre méthode et construire des liens entre le nœud la méthode courantes et les nœuds des méthodes invoquées
  end for
end for

5.5.3 Imprécision de l'analyse insensible au contexte

L'analyse des flux de contrôle permet de déterminer dans une suite d'exécution, quelle sera la prochaine instruction qui va être exécutée. Cependant, les variables pointées par l'instruction peuvent dépendre fortement du contexte de la méthode dans laquelle elles sont traitées. Dans une analyse insensible au contexte, deux invocations de contextes différents à une méthode `m` ne sont pas distinguées. Les arguments formels des méthodes vont pointer sur tous les objets passés dans des méthodes de différents appels. De façon similaire, le résultat de retour d'une méthode `m` pointera sur tous les objets qui peuvent être retournés par `m`. Ce problème est appelé "le chemin non réalisable" [LL05]. Afin de mieux comprendre ce problème considérons l'exemple suivant :

Listing 5.1– Sensibilité au contexte

```
1 class MyClass{
2     private String str;
3     public MyClass(String str)}
4         this.str=str;
5     }
6     public String getSTR(){
7         return this.str;
```

```
8       }
9   }
10  public void doGet(HttpServletRequest request, HttpServletResponse
        response) throws IOException {
11
12
13  String UserInput=request.getParameter("input");
14
15  MyClass Obj1= new MyClass(UserInput);
16  String X="Exemple";
17  MyClass Obj21= new MyClass(X);
18
19
20  String S1=Obj1.getSTR();
21  String S2=Obj2.getSTR();
22  }
```

Exemple Supposons le code du *Listing 5.1.* La classe `MyClass` agit comme un encapsuleur de la chaine de caractères `str`. Le code crée deux objets, *Obj1* et *Obj2*, instances de la classe *MyClass*, et appelle la méthode `getSTR()` à partir de ces deux objets. L'analyse insensible au contexte consiste à fusionner les informations des deux appels à `getSTR( )` dans les lignes 20 et 21. La variable `this` de la ligne 7, pointera sur les objets obj1 et obj2 alloués en ligne 15 et 17. Ainsi `this.STR` pointera soit sur l'objet retourné en ligne 13 ou la chaine de caractères «Exemple» de la ligne 16. Au final, si on considère que l'objet retourné dans la ligne 13 est étiqueté, alors en appliquant la technique de dépendance causale, les deux variables `S1` et `S2` seront considérées toutes les deux comme étiquetées. Une analyse sensible au contexte serait plus précise : la méthode `getSTR()` serait analysée deux fois dans deux contextes différents, et seule la variable `S1` serait étiquetée.

5.6 Analyse des flux de données

Notre analyse est basée sur le contrôle des flux de données afin de suivre la propagation des données dans tout le programme. Nous nous intéressons à la fois aux flux de données internes à une méthode (analyse intra-procédurale) et aux flux échangés entre différentes méthodes de l'application (analyse inter-procédurale). L'analyse intra-procédurale consiste à suivre la propagation des données dans une méthode, en tenant compte des différents types d'instructions, des données manipulées et de cas particuliers comme les branchements et les exceptions, etc. L'analyse inter-procédurale permet de suivre les données propagées entre différentes méthodes à travers les paramètres ou la valeur de retour d'une méthode ou dans les champs de classe communs à un ensemble de méthodes de cette classe.

5.6.1 La dépendance causale

Notre approche applique une analyse de flux de données, basée sur la technique de dépendance causale (Chapitre 4, section 4.3). Afin de suivre la propagation des données dans l'exécution

abstraite d'un programme, nous définissons une étiquette *isTainted* associée à chaque *item* (élément d'une pile d'opérande, ou de la table des variables locales) potentiellement malicieux dans les structures de données composant la JCVM simulée.

La dépendance causale est initiée par notre module *XSSDetector* qui associe à chaque *item* correspondant à un état source S de l'automate XSS une étiquette *isTainted*. L'étiquette est propagée à tous les items générés à partir d'un état D. Une vulnérabilité est détectée quand une donnée étiquetée est référencée par une fonction correspondant à un état final F de l'automate.

Findbugs permet de faire une analyse abstraite de l'exécution du *bytecode*. L'analyse qu'il propose est intra-procédurale. Elle traite tous les cas de branchement conditionnel et les exceptions. Cependant, de nombreux cas comme le suivi des flux externes à la méthode (variables de classes, champs statiques) et le traitement des conteneurs (tableau et vecteur) ne sont pas prévus. D'autre part *Findbugs* parcourt toutes les méthodes de l'application sans exception, indépendamment de leur contexte (*algorithm 1*).

Notre outil commence son analyse à partir d'un point d'entrée du programme. Généralement une servlet commence son exécution à partir d'une des méthodes `doGET()` ou `doPost()` qui traitent respectivement les méthodes HTTP GET et POST. Seules les méthodes invoquées à partir d'un point d'entrée sont analysées. Les méthodes déclarées qui n'ont jamais été invoquées dans le programme ne sont pas vérifiées, ce qui permet de réduire considérablement le nombre de faux positifs qu'aurait générées *Findbugs*.

Afin de mieux comprendre le principe de la dépendance causale, supposons le code ci dessous, le code du *listing 5.3* représente le *bytecode* correspondant au code source du *listing 5.2*

Listing 5.2– Code source

```
1
2  public void doGet(HttpServletRequest request, HttpServletResponse
       response) throws IOException {
3          response.setContentType("text/html");
4          PrintWriter out = response.getWriter();
5             try {
6
7                 String X;
8                 X=request.getParameter("name");
9
10                out.println(X);
11
12              } finally {
13                 out.close();
14         }
15     }
```

Listing 5.3– Bytecode

```
1  Code(max_stack = 2, max_locals = 7, code_length = 48)
2  ...
3  15:   aload_1
4  16:   ldc               "name" (5)
5  18:   invokeinterface   javax.servlet.http.HttpServletRequest.
       getParameter(Ljava/lang/String;)Ljava/lang/String;          (6)
         2      0
6  23:   astore            %5
7  25:   aload_3
8  26:   aload             %5
9  28:   invokevirtual     java.io.PrintWriter.println (Ljava/lang/
       String;)V (7)
10 ...
11 47:   return
```

Dans la ligne 3 du *bytecode*, `aload-1` correspond au chargement de la variable locale 1 dans la pile des opérandes. Dans la ligne 4, la référence à la constante «name » retrouvée à partir du *constant pool* est empilée dans la pile des opérandes.

À la ligne 5 la méthode `HttpServletRequest.getParameter()` est invoquée. Cette méthode permet de retourner des données entrées par l'utilisateur. L'interprétation de l'instruction `invokevirtual` consiste à dépiler les deux items au sommet de la pile et à empiler l'item correspondant à la valeur retournée par la méthode `HttpServletRequest.getParameter()`. *XSSDetector* détecte que l'appel à la méthode `HttpServletRequest.getParameter()` correspond au passage à l'état S de l'automate XSS, dans ce cas une étiquette *isTainted* sera associée à l'item ajouté à la pile.

À la ligne 6, l'instruction `astore` consiste à retirer le sommet de la pile et le charger dans la table des variables locales à l'index indiqué. Dans ce cas, l'étiquette est également propagée à la table des variables locales en associant l'étiquette *isTainted* à l'item chargé dans la table des variables locales.

L'appel à l'instruction `out.println(X)` du code source correspond aux instructions 25, 26 et 28 du *bytecode*. L'interprétation de l'instruction `aload_3` consiste à charger l'item de la variable locale 3 dans la pile. cet item correspond au paramètre de la méthode `println()`. Vu que cette variable locale est étiquetée alors l'item chargé au sommet de la pile va être aussi étiqueté. L'interprétation de l'instruction `invokevirtual` (ligne 9) consiste à dépiler le sommet de la pile. `XSSdetector` vérifie d'abord si cet item est étiqueté. Ainsi, puisque c'est le cas et que la méthode invoquée `JAVA.IO.PrintWriter.println()` est une des méthodes définies dans l'état F de l'automate XSS, alors *XSSdetector*, détecte une correspondance complète avec l'automate XSS et lance un avertissement pour signaler une vulnérabilité XSS (volatile dans ce cas).

Afin de faciliter la maintenance du programme, l'ensemble du chemin parcouru entre un état S et l'état final F est sauvegardé dans un fichier de journalisation.

5.6.2 Étiquetage des conteneurs

La plupart des outils d'analyse basés sur la dépendance causale effectue un étiquetage d'un objet si un de ses éléments est détecté comme potentiellement dangereux (étiqueté). Cependant

cette technique peut générer un nombre important de faux positifs ou de faux négatifs. En effet, si une donnée malicieuse contenue dans l'objet est nettoyée de tout caractère malicieux, cette dernière ne devrait plus être considérée comme malicieuse. Dans ce cas, faut-il annuler l'étiquette de l'objet la contenant ?

La réponse à cette question pose un problème dans les deux cas. L'annulation de l'étiquette présente le risque de faux négatifs, car d'autres éléments du conteneur peuvent être malicieux et la garder risque de générer des faux positifs si tous les autres éléments du conteneur sont correctement filtrés.

Algorithm 3 étiquetage des conteneurs (tableaux et vecteurs)

```
1: container[ ] ;
2: String str ;
3: ...
4: /* étiquetage d'un tableau */
5: if instruction de chargement de str dans container[ ] then
6:    construire un tableau des indices tableauDesIndices[ ] associé au conteneur
7:    if str est étiqueté then
8:       étiqueter la case correspondante dans le tableau des indices tableauDesIndices[ ]
9:       étiqueter l'objet container[ ]
10:   end if
11: end if
12:
13:
14: /* annulation de l'étiquetage en cas de filtrage */
15:
16: boolean bool=faux ;
17:
18: if container[i] est filtré then
19:    if tableauDesIndices[i] est étiqueté then
20:       tableauDesIndices[i].isTainted=false ;
21:    end if
22:    for int j allant de zéro à tableauDesIndices[ ].length-1 do
23:       if tableauDesIndices[i] est étiqueté then
24:          bool=vrai ;
25:       end if
26:    end for
27:
28:    if bool==vrai then
29:       étiqueter container[ ] ;
30:    end if
31: end if
```

Dans Java Card 3, les conteneurs supportés sont les tableaux et les vecteurs. Afin de palier au problème de dépendance causale dans ce cas, nous avons choisi d'étiqueter les éléments du conteneur aussi. Pour cela, à chaque objet conteneur (tableau ou vecteur) déclaré dans l'application, l'analyseur définit un tableau correspondant aux indices du conteneur (*Algorithm 3*). Quand une donnée malicieuse est chargée dans le conteneur, en plus de l'étiquetage de l'objet conteneur, l'élément du tableau des indices, correspondant à la donnée non fiable est également étiqueté. Si un élément du conteneur est filtré, l'étiquette de l'indice lui correspondant est retirée. L'étiquette du tableau est retirée si et seulement si aucun des éléments du tableau des indices n'est étiqueté.

Étiquetage des objets instances de classe

Dans cette section, nous désignons par objet, une instance de classe. De même que pour les tableaux et les vecteurs, l'étiquetage des objets instances d'une classe pause un problème.

Pour comprendre la nécessité de l'étiquetage des objets. Supposons l'exemple du *listing 5.4*. L'objet `obj1` est une instance de la classe *MyClass* et un de ses attributs `attribut1` est étiqueté. Supposons aussi un tableau `tab` des objets instances de la classe *MyClass*. Ce tableau est persistant car il est défini par le modificateur `static`. L'objet *obj1* est stocké dans `tab1[ ]`.

Si l'analyse ne prévoit pas l'étiquetage des objets dont un des attributs est étiqueté alors `obj1` ne sera pas étiqueté, et par conséquent, le tableau `tab1[ ]` ne le sera pas non plus. L'analyse ne pourra donc pas détecter la présence de l'attribut `attribut1` dans un champ persistant. La donnée malicieuse serait ainsi stockée de manière persistante dans la carte, ce qui correspond à une vulnérabilité XSS persistante.

Afin de palier à ce cas, nous avons choisi d'étiqueter tout objet dont un des attributs est étiqueté. Si l'attribut `attribut1` est neutralisé par un filtrage et si aucun des autres attributs de l'objet n'est étiqueté alors l'étiquette de l'objet est annulée.

Listing 5.4– Les objets instances de classe

```
1
2   class MyClass{
3
4      private String attribut1;
5      private String attribut2;
6
7      public MyClass(String str){
8          this.attribut1=attribut1;
9          this.attribut2=attribut2;
10     }
11  }
12  ...
13  static MyClass[] tab1;
14
15  public void doGet(HttpServletRequest request, HttpServletResponse
        response) throws IOException {
16
17     ...
18
19     String UserInput=request.getParameter("input");
20     String x="str";
21
22     MyClass Obj1= new MyClass(UserInput,x);
23
24     tab1.addelement(obj1);
```

```
25         ...
26 }
```

5.6.3 Détection des champs persistants

Rappelons que dans Java Card 3, un champ est dit persistant (sauvegardé en mémoire non volatile) s'il est défini par le modificateur `static`, ou qu'il est instance d'une des classes `javacard.framwork.applet` ou `javax.Servlet.ServletContext` . Un champ volatile peut devenir persistant par accessibilité, si un champ persistant lui est affecté. Afin de détecter des attaques XSS non volatiles, il faut vérifier qu'un champ persistant n'est pas étiqueté par l'étiquette *isTainted*. Pour implémenter cette propriété, nous avons défini une nouvelle étiquette *isPersistent* qui est associée à tout champ persistant et qui est propagée à tout item ou champ dépendant d'un champ étiqueté. Une vulnérabilité XSS persistante est détectée en présence d'un item doublement étiqueté par les étiquettes *isTainted* et *isPersistent*.

5.6.4 Dépendance causale inter-méthodes

Une méthode peut avoir une valeur de retour qui peut être le résultat de la manipulation de données potentiellement malicieuses (étiquetées). La valeur de retour peut être utilisée dans une autre méthode, d'une même classe ou d'une classe différente. D'autre part, un paramètre d'une méthode peut être le résultat de la manipulation de données étiquetées.

Notre analyse de flux de données est inter-procédurale, permettant de propager l'étiquette entre une méthode appelante et une méthode appelée. La *figure 5.4* illustre le fonctionnement de la propagation de l'étiquette.

Supposons que `M1` est une méthode qui invoque une méthode `M2`. La méthode `M2` prend en entrée deux paramètres, ces derniers sont calculés dans la frame `frame1` de la méthode `M1`. Notre analyse permet de vérifier si les paramètres calculés dans `frame1` sont étiquetés et propage l'étiquetage aux items correspondants dans la table des variables locales de `M2`. Ce processus est réalisé comme suit :

- Sauvegarde du contexte de la méthode `M1` et de l'état de la frame `frame1` ;
- Création d'une nouvelle frame `frame2` pour la méthode `M2` ;
- Si les paramètres calculés dans `frame1` sont étiquetés alors, mettre à jour les items de la table des variables locales de `frame2`.

À la fin du traitement de `M2`, si cette dernière retourne une valeur, un item correspondant à cette valeur sera empilé dans la pile de `frame2`. Une copie de cet `item` est empilée sur la pile d'exécution de la frame `frame1`. Ainsi, si cet item est étiqueté, l'étiquette sera propagée à `frame1`. A présent, l'analyse de la méthode M2 est terminée. La frame `frame2` est détruite et notre analyseur retourne au contexte de la méthode `M1` sauvegardé pour continuer le traitement des instructions suivantes.

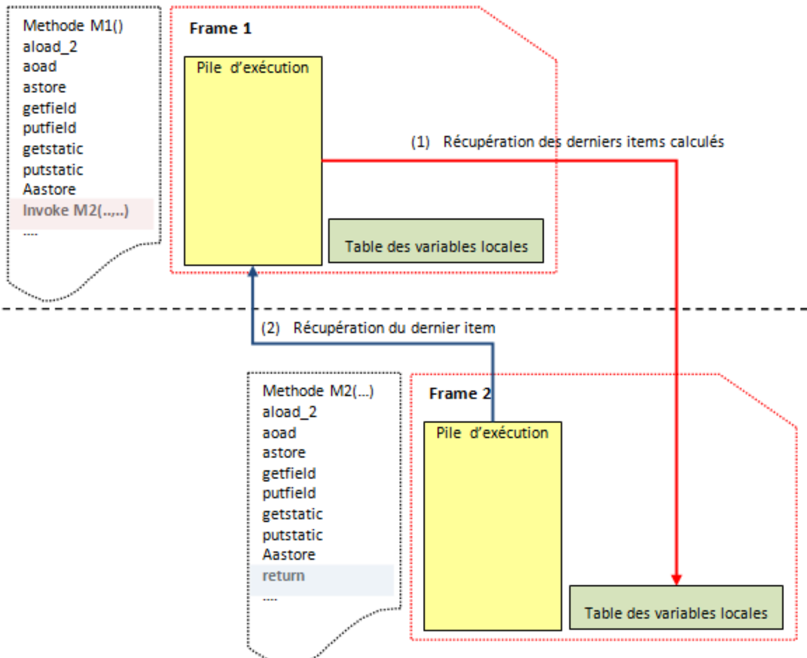

Fig. 5.4 – Suivi des flux de données inter-méthodes

5.7 API de filtrage XSS pour des applications cartes à puces

Afin de protéger une application contre des attaques XSS, il faut considérer toute donnée générée à partir d'une source externe à l'application comme potentiellement dangereuse. La meilleure protection contre des attaques par injection de code est de vérifier et d'échapper tous les caractères spéciaux contenus dans ces données avant leur utilisation dans un contexte critique où ils pourraient avoir une autre interprétation que celle attendue.

Contrairement à d'autres langages comme PHP, le langage Java ne prévoit pas de librairies ou de fonctions permettant de vérifier et de neutraliser des données potentiellement malicieuses. Cependant, des tierces parties proposent des API que le développeur peut utiliser dans le développement de son application. Parmi ces API, OWASP propose l'API ESAPI [ESA] qui comporte entre autres, des fonctions de filtrage bien définies contre des attaques XSS. ESAPI est proprement développée et son filtrage est efficace. Elle est également simple d'utilisation. Cependant, elle ne correspond pas à notre plateforme d'étude : la Java Card 3. Elle utilise des librairies Java qui ne sont pas supportées dans Java Card 3 telles que *java.util.HashMap, java.util.LinkedList* , etc.

Notre API *JCXSSFilter* est une réécriture adaptée en Java Card 3 de la partie de ESAPI qui permet de valider et d'encoder des données potentiellement dangereuses. Le filtrage proposé est basé sur le modèle de "liste blanche" qui refuse tout ce qui n'est pas spécifiquement autorisé.

Compte tenu de la façon dont les navigateurs Web analysent du code HTML, chaque point d'insertion a des règles de sécurité différentes. Souvent, la sécurité utilisée consiste à encoder en entités HTML tous les caractères spéciaux. Cette mesure n'est efficace que dans des cas de données injectées dans le corps HTML, au niveau de balises HTML simple telle que <div>. Un filtrage spécifique est nécessaire pour d'autres points d'insertion tels que des blocs de *JavaScript*, les valeurs

d’attributs HTML, dans des balises de style CSS, ou dans des attributs d’URL [OWAb]. Nous présentons dans ce qui suit chacun de ces points d’insertion et le filtrage spécifique utilisé pour chacun.

5.7.1 Filtrage des données insérées dans un élément HTML

Les données non fiables peuvent être placées dans un corps HTML. Cela inclut l’intérieur des balises simples comme `div, p, b, td`, etc. Pour éviter que ces données passent d’un contexte de données à un contexte de code, il suffit de faire un filtrage simple qui consiste à échapper certains caractères pour éviter qu’ils soient interprétés comme un code ou un script par le navigateur Web.

Exemple : points d’insertion dans des valeurs d’attributs HTML

```
<body>...filtrer les données avant de les placer ici...</body>
```

```
<div>...filtrer les données avant de les placer ici...</div>
```

Échapper ces caractères consiste à utiliser un encodage en entités HTML pour empêcher le passage à tout contexte d’exécution, tels que le script, le style, ou des gestionnaires d’événements. Utiliser la représentation hexadécimale est également recommandé dans les spécifications. En plus des 5 caractères importants en XML (&, <,>, ", ’), la barre oblique est incluse car elle contribue à mettre fin à une entité HTML.

Caractère	**Encodage en entité HTML**
&	& ;
<	< ;
>	> ;
"	" ;
’	' ;
/	/ ;

Table 5.1 – Exemples de caractères spéciaux et leur encodage en entités HTML

5.7.2 Filtrage des données insérées dans des attributs HTML

Les données non fiables peuvent être insérées dans les valeurs d’attributs comme la largeur, le nom, la valeur, etc. Elles sont utilisées dans les attributs complexes comme `href, src`, ou des gestionnaires d’événements comme `onmouseover`.

Les valeurs d’attributs peuvent être entourées de double côtes, de côtes simples, ou parfois sans aucune côte.

Exemple : points d’insertion dans des valeurs d’attributs HTML

```
<div attr=...filtrer les données avant de les insérer ici...>content</div>
```

```
<div attr='...filtrer les données avant de les insérer ici...'>content</div>
```

```
<div attr="...filtrer les données avant de les insérer ici...">content</div>
```

Pour prévenir la commutation des attributs HTML et empêcher que des valeurs de données sortent du contexte de l'attribut pour lesquelles elles sont prévues, il faut encoder tous les caractères spéciaux par le format &#xHH, H étant la représentation en hexadécimale d'un caractère, et le caractère & est utilisé pour spécifier le début d'un nouveau caractère.

Une côte fait partie des caractères spéciaux qu'il faut échapper car elle marque la fin d'une valeur d'un attribut côtée. Le caractère & est également spécial car il introduit un caractère ou sépare des paramètres. Cependant souvent les développeurs laissent les attributs HTML sans côtes. Les attributs non côtés peuvent être mal interprétés (sortir du contexte prévu) et répartis par des caractères dont , / ; < = > et | ^. Afin de rendre le filtrage puissant, nous avons suivi les recommandations d'OWASP, et nous avons appliqué un encodage d'un ensemble large de caractères qui comprend tous les caractères de valeurs ASCII inférieur à 256, exceptés les caractères alphanumériques en leur format \&#xHH.

5.7.3 Filtrage des données insérées dans un code *JavaScript*

Le troisième point d'insertion est au niveau des codes *JavaScript* générés dynamiquement à partir de blocs de scripts ou des attributs de gestionnaires d'événements. Le seul endroit sûr où des données non fiables peuvent être insérées dans un code est dans un point d'insertion entre deux côtes. L'insertion de données non fiables dans d'autres contextes *JavaScript* est dangereux. En effet, il est très simple de changer le contexte d'exécution courant en utilisant des caractères dont ;, =, +, etc. Il est donc recommandé de les utiliser avec prudence et appliquer un filtrage puissant pour prendre en compte les différents cas possibles.

Exemple : points d'insertion dans un script

```
<script>alert('...filtrer les données avant de les insérer ici...')</script>

<script>x='...filtrer les données avant de les insérer ici...'</script>

<div onmouseover="x='...filtrer les données avant de les insérer ici...'"</div>
```

La spécification OWASP déconseille fortement le filtrage utilisant des raccourcis comme un antislash. En effet, le caractère «côte» peut être lu par l'analyseur d'attributs HTML qui est toujours exécuté en premier. D'autre part, ce type de filtrage est vulnérable à une attaque par échappement d'un échappement qui consiste à utiliser un deuxième antislash pour annuler celui utilisé en protection. Si l'attaquant injecte la chaîne \" et que la protection appliqué utilise un antislash \ alors on obtiendra la chaine \\" où la protection est annulée et la côte est interprétée.

Le filtrage que nous avons implémenté consiste à échapper tous les caractères de valeurs ASCII inférieur à 256, exceptés les caractères alphanumériques, en les encodant au format \xHH afin de prévenir tout passage d'un contexte de valeur de donnée à un contexte de script ou d'un autre

attribut. Nous avons appliqué l'encodage sur un ensemble aussi large de caractères, car les valeurs d'attributs gestionnaires d'événements sont souvent laissés sans côtes. Les attributs non côtés peuvent être mal interprétés et éclatés en plusieurs parties par des caractères tels que `[espace] % * + , - / ; < = > ^`. D'autre part, la balise de fermeture `</script>` pourrait fermer un bloc de script même si elle est injectée à l'intérieur d'une chaîne de caractères entre côtes car l'analyseur HTML est exécuté avant l'analyseur *JavaScript*.

Cependant dans certains cas, les fonctions *JavaScript* ne devraient jamais utiliser des données non fiables comme entrées, même si ces données sont filtrées. Par exemple :

```
<script>
 window.setInterval('...Quel que soit le filtrage utilisé une vulnérabilité
                     XSS est possible ici...');
</script>
```

5.7.4 Filtrage et validation des données insérées dans les propriétés de style HTML

Les balises de style CSS sont utilisées pour modifier la typographie du texte. Elles peuvent contenir des points d'insertions où le développeur peut définir une propriété d'un style. Cependant, CSS est très puissant, et peut être exploité pour de nombreuses attaques. Par conséquent, il est important d'utiliser des données non fiables uniquement dans une valeur de propriété et pas dans d'autres endroits dans les données de style.

Exemple : points d'insertion dans des propriétés CSS

```
<style>selector { property : ...Filtrer les données non fiables
                avant de les insérer ici...; } </style>

<style>selector { property : "...Filtrer les données non fiables
                avant de les insérer ici..."; } </style>

<span style="property : ...Filtrer les données non fiables avant
               de les insérer ici...">text </style>
```

Certains contextes CSS ne devraient jamais utiliser des données non fiables comme entrée même si un échappement correct de CSS est prévu. Il est recommandé d'éviter des propriétés complexes comme les URL, les propriétés de comportement définies dans Internet Explorer. Il est donc recommandé de vérifier que les URL commencent par "http" et non pas par "javascript" et que les propriétés ne commencent jamais par "expression" (expression() est définie par IE : elle permet d'utiliser un code *JavaScrip* comme valeur d'une propriété CSS)
Exemple :

```
{ background-url : "javascript:alert(1)"; }
{ text-size: "expression(alert('XSS'))"; }
```

Pour les mêmes raisons que le filtrage précédent, l'utilisation d'un échappement par le caractère antislash `"\"` est très déconseillée dans les recommandations d'OWASP. Le filtrage proposé consiste à encoder tous les caractères ASCII inférieurs à 256, exceptés les caractères alphanumériques, au format `\HH`. Cet encodage est suffisamment fort pour empêcher une XSS lorsque les données non fiables sont placées dans des contextes sans côtes. Les valeurs d'attributs non côtées peuvent être réparties si elles contiennent des caractères spéciaux dont `[espace] * % + , - / ; < = > | ^`. D'autre part, l'injection de la balise de fermeture `</style>` pourrait fermer un bloc de style, même dans le cas où elle est utilisée à l'intérieur d'une chaîne entre double côtes car l'analyseur HTML s'exécute avant l'analyseur *JavaScript*.

5.7.5 Filtrage des données insérées dans des valeurs de paramètres d'URL

Le dernier point d'insertion que nous avons pris en compte est la valeur d'un paramètre d'une URL. Une URL peut être utilisée dans une requête HTTP GET. Ses paramètres doivent être considérés comme données non fiables et être filtrés en conséquence.

Exemple : points d'insertion dans des attributs d'URL

```
<a href="http://www.somesite.com?test=...Filtrer les données
        non fiables ici...">link</a >
```

Les attributs d'URL non côtés peuvent sortir de leur contexte par de nombreux caractères spéciaux dont `[espace] *% +, - / ; < = > | ^`. Le filtrage dans ce cas consiste à encoder au format `%HH` tous les caractères de valeur ASCII inférieures à 256, exceptés les caractères alphanumériques.

Cependant, dans le cas d'une URL complète, il est recommandé de la valider avant de l'encoder. Si une entrée non fiable est destinée à être placée dans `href, src` ou d'autres attributs d'URL, elle devrait être validée afin de s'assurer qu'elle ne pointe pas vers un protocole inattendu, particulièrement des liens *JavaScript*. L'URL devrait être ensuite encodée en fonction du contexte d'affichage comme n'importe quel autre type de données HTML. Par exemple, un développeur appliquant une URL axée sur les liens `href` doit utiliser un encodage des attributs.

```
String userURL = request.getParameter( "userURL" )
userURLvalide= Valider(userURL) /* valider la valeur de userURL */
if (isValidURL) {
    <a href="<\%=utiliser le filtrage des atributs HTML
             sur userURLvalid\%>">link</a>
 }
```

5.8 Conclusion

Notre outil d'analyse de *bytecode* est un moyen de prévention contre des vulnérabilités XSS permettant de vérifier qu'une application est bien développée, utilisant notre API de filtrage *JCXXS-Filter* dans tout point d'insertion de données non fiables. Cette analyse est effectuée avant le chargement de l'application dans la carte à puce. Notre outil effectue une analyse statique et se

base sur la simulation du processus d'exécution d'un *bytecode* et du comportement de l'interpréteur de la JCVM. Il utilise la technique de dépendance causale dans l'analyse des flux de données intra- et inter-procéduraux et effectue un suivi de la propagation des données dans toute l'application en se basant sur un graphe d'appel interclasses sensible au contexte. Afin de réduire le nombre de faux positifs, nous avons géré les conteneurs et les données qu'ils contiennent de façon précise afin d'éviter qu'une donnée soit considérée « à tort » comme non fiable. L'implémentation de cet outil est présentée dans le chapitre suivant.

Chapitre 6

Implémentation et expérimentations

Sommaire

6.1 Présentation de BCEL **85**
6.2 Interprétation abstraite de la JCVM **86**
6.2.1 Simulation d'une frame 86
6.2.2 Représentation d'un item 88
6.2.3 Plugin XSSDetector 90
6.3 Implémentation de *JCXSSFilter* **91**
6.4 Expérimentations **91**
6.4.1 Stratégie d'évaluation de l'outil d'analyse statique 92
6.4.2 Temps de l'analyse 93
6.4.3 Détection de vulnérabilités 94
6.4.4 Faiblesse de l'analyse statique 97
6.4.5 Surcoût du filtrage 100
6.5 Conclusion **101**

Notre outil d'analyse statique fournit à son utilisateur la liste des points du programme où une vulnérabilité XSS est détectée. L'utilisateur doit par la suite corriger manuellement ces failles avant de charger l'application dans la carte à puce. Cet outil a été implémenté en Java en utilisant l'API *ByteCode Engineering Library*. Notre API de filtrage *JCXSSFilter* a été développée en Java Card 3 (sous ensemble de Java).

6.1 Présentation de BCEL

BCEL [Apa06] est une librairie open source permettant d'analyser, de créer, et de manipuler des fichiers `class`. Ces fichiers contiennent des classes qui sont chargées en mémoire sous forme d'objets contenant toutes les informations sur la structure de cette classe : les méthodes, les attributs et les instructions (*Opcode*). Elle est composée principalement de deux paquetages Java :

1. le paquetage contenant des classes permettant de lire le *bytecode*. Il est particulièrement utilisé dans l'analyse des applications Java dont le code source n'est pas fourni. La principale structure de données de cette classe est la classe `JavaClass` qui contient les méthodes, les champs, le pool de constantes de la classe, etc.
2. le paquetage Java contenant des classes permettant de modifier ou de générer des objets de type `JavaClass` ou `Method`.

L'outil que nous avons développé durant cette thèse et l'outil *Findbugs* sur lequel se base notre analyse utilisent cette librairie pour lire et parcourir les fichiers `class` d'une application.

6.2 Interprétation abstraite de la JCVM

Tel que présenté dans le chapitre précédent, notre analyse statique se base sur une interprétation abstraite de l'exécution d'un programme. L'interpréteur de la machine virtuelle est simulé en implémentant le dispositif de frame d'une méthode, la table des variables locales et la pile des opérandes. L'espace mémoire partagé par un ensemble de méthodes est également simulé, afin de gérer la présence des variables de classe et des champs statiques dans un programme.

La simulation ne se limite pas à implémenter les structures de données manipulées par l'interpréteur, mais elle tient compte également du fonctionnement de ce dernier pour l'ensemble des instructions possibles dans un *bytecode*. Notre analyse utilise un graphe d'appel interclasses, sensible au contexte et parcourt tous les chemins d'exécution possibles du programme analysé. Le parcours du graphe d'appel commence à partir de points d'entrées bien définis correspondant à l'appel à des méthodes de traitement de requêtes HTTP définis dans la classe `javax.servlet.http.HTTPServlet` (exemple : doGet() et doPost()). Les structures de données composant la JCVM sont mises à jour instantanément, suivant le type d'instructions visitées et le type de données manipulées.

6.2.1 Simulation d'une frame

L'interpréteur abstrait est principalement implémenté dans la classe `opcodeStack` de *Findbugs*. Il prend en entrée un fichier `class` qui est interprété et sauvegardé dans un objet Java. Il effectue une exécution abstraite des méthodes en simulant le déplacement du pointeur de code entre les différentes instructions et la frame Java. Nous avons modifié la classe `opcodeStack` pour introduire la dépendance causale (étiquetage des items) et pour compléter l'implémentation de quelques instructions (*opcode*) qui n'étaient pas complètement gérées. Le code du *listing 6.1* nous montre l'implémentation des structures de données représentant une frame avec sa pile des opérandes et sa table des variables locales.

Listing 6.1– Abstraction d'une frame

```
1 class opcodeStack{
2     private List<Item> stack;
3     private List<Item> lvValues;
4
```

```
5       /* toutes les methodes necessaires a la gestion de la pile et
           de la table des variables locales: Item  pop(), void
           replaceTop(Item newTop), void pop(int count),  void push(
           Item i), etc. */
6
7
8       public void sawOpcode(int seen) {
9
10      /*seen est la representation en decimal d'une instruction (
           opcode) */
11
12          switch(seen){
13
14              /* traitement de l instruction et  mise a jour de la
                   pile et la tables des variables locales suivant le
                   type d instruction.*/
15
16          }
17      }
18  }
```

La pile et la table des variables locales sont représentées par des structures de données de type `list` (lignes 2 et 3) contenant des objets de type `Item`. La méthode `sawOpcode(int seen)` est appelée à chaque analyse d'une nouvelle instruction du bytecode ; son paramètre `seen` est une représentation en entier d'un *Opcode* visité. Cette méthode implémente le fonctionnement de l'interpréteur (ajouter ou supprimer un item de la pile ou de la table des variables locales ou de la mémoire partagée) pour tous les cas possibles d'instructions.

L'exécution abstraite de *bytecode* se base sur le suivi du graphe d'appel interclasses de l'application. Elle consiste à maintenir le pointeur du code et la frame de chaque méthode de l'application. Pour chaque méthode visitée, l'interpréteur crée une frame composée principalement de la pile et de la table des variables locales. La hauteur de la pile va augmenter ou diminuer en fonction des instructions pointées. La représentation des structures de l'interpréteur abstrait est la même que l'interpréteur classique de la machine virtuelle. Cependant, l'interprétation des `opcodes` est différente. L'interpréteur abstrait effectue une analyse statique `off-card` [15] du *bytecode* ; les instructions ne sont pas réellement exécutées, mais une abstraction de leur exécution est effectuée ; par exemple, dans le cas de l'instruction `iadd`, la hauteur de la pile est diminuée de deux éléments (correspondant aux opérandes) puis augmentée d'un élément représentant le résultat de l'addition mais l'opération d'addition n'est pas réalisée.

D'autre part, dans le cas des branchements conditionnels, la vérification de la condition étant impossible, nous traitons donc les différents chemins possibles que la condition soit vraie ou fausse. Ceci est le cas des exceptions aussi.

15. réalisé à l'extérieur de la carte à puce

6.2.2 Représentation d'un item

Les objets `Item` sont représentés par une structure de données qui contient différentes informations sur une donnée stockée dans la pile ou la table des variables locales (*listing 6.2*).

Listing 6.2– Représentation d'un item

```
1  class Item{
2
3          /* ensemble des information representant un item*/
4
5          private int pc = -1;
6          private String signature;
7          private Object constValue = UNKNOWN;
8          private @CheckForNull ClassMember source;
9          private int registerNumber = -1;
10         private Object userValue = null;
11         private int fieldLoadedFromRegister = -1;
12         public static final @SpecialKind;
13         int isTainted = 0;
14         int UnTainted = 15;
15         public HttpParameterInjection injection = null;
16         public tableauDesIndices[]=null;  /* utiliser dans le cas
              ou l item correspond a un tableau ou un vecteur*/
17
18         /* . . . */
19
20         public Item(Item it) {
21             this.signature = it.signature;
22             this.constValue = it.constValue;
23             this.source = it.source;
24             this.registerNumber = it.registerNumber;
25             this.userValue = it.userValue;
26             this.injection = it.injection;
27             this.flags = it.flags;
28             this.specialKind = it.specialKind;
29             this.pc = it.pc;
30
31          }
32
33 /* Toutes les fonctions permettant de manipuler les items ou de
       recuperer des informations: Item merge(Item i1, Item i2),
       getFieldLoadedFromRegister(), getPC(), etc. */
34
35         public boolean isServletParameterTainted() {
```

```
36              return getSpecialKind()==Item.isTainted;
37          }
38
39          public void setServletParameterTainted() {
40               setSpecialKind(Item.isTainted);
41          }
42
43          public void setServletParameterUnTainted() {
44               setSpecialKind(Item.UnTainted);
45          }
46
47          public void setSpecialKind(int SpecialKind) {
48               item.SpecialKind=SpecialKind;
49          }
50     }
```

Étiquetage des items. L'attribut `SpecialKind` est utilisé afin d'appliquer l'étiquetage des items malicieux. Un item est source d'une donnée extérieure au programme si `SpecialKind` est égal à `isTainted`. Cet attribut est manipulé par les fonctions suivantes :

- `setServletParameterTainted()` est appelée pour étiqueter les items résultant d'un état S de l'automate XSS (chapitre 5, section 5.2), ou pour propager l'étiquette à un item généré à partir d'un état D,
- `isServletParameterTainted()` vérifie si un item est étiqueté ou pas,
- `setServletParameterUnTainted()` permet d'annuler l'étiquette d'un item ; elle est appelée dans le cas où l'item correspondant passe en paramètre d'une fonction de filtrage.

L'attribut `tableauDesIndices[]` est utilisé dans le cas où l'item correspond à un objet de type tableau ou vecteur. Ses éléments sont de valeur `"isTainted"` ou `"UnTainted"`. À chaque fois qu'un élément est ajouté dans un tableau à une position `i`, si cet élément est étiqueté alors la valeur `tableauDesIndices[i]` est mise à `"isTainted"` sinon elle est mise à `"UnTainted"`.

L'attribut `injection` est un objet instance de la classe `HttpParameterInjection`. Il est utilisé pour sauvegarder un ensemble d'informations sur l'origine et le chemin d'exécution qui a mené jusqu'à la vulnérabilité. À chaque visite d'un état D, l'objet `injection` de l'item correspondant est mis à jour en sauvegardant le numéro de ligne courant du code source et le nom de la méthode visitée.

Listing 6.3– La classe HttpParameterInjection

```
1 public class HttpParameterInjection {
2     private String parameterName;
3     private int line;
4     private Vector <LineInMethod> path=new Vector();
5     private int pc;
6
```

```
7       public HttpParameterInjection(String parameterName, int pc,
           Method m, Vector path) {
8
9           this.parameterName = parameterName;
10          this.path=path;
11          int l=getSourceLine(pc, m);
12          LineInMethod line=new LineInMethod(l,m);
13          this.path.add(line);
14          }
15
16      /* Calculer le numero de ligne dans le code source a partir de
           pc */
17
18      private int getSourceLine(int pc, Method m){
19
20      }
21
22      private class LineInMethod{
23          public int line;
24          public Method method;
25
26          public LineInMethod(int line,Method m){
27              this.line=line;
28              this.method=m;
29           }
30      }
31
32  }
```

6.2.3 Plugin XSSDetector

Le plugin *XSSDetector* se charge de :
– l'étiquetage des items générés dans un état S ;
– la détection des états F ;
– la propagation des étiquettes inter-méthodes.

Au lancement de `Findbugs` le plugin *XSSDetector* est chargé. Les fichiers `class` sont transformés en `JavaClass`. L'analyse commence par la construction du graphe d'appel complet de l'application à partir du module *inter-class-calls*. *XSSDetector* parcourt ensuite le *bytecode* à partir des méthodes qui traitent des requêtes HTTP (doGet(), doPost(), etc.) et crée une instance de la classe `opcodeStack`(que nous avons améliorée) pour chaque méthode visitée qui génère la frame de cette méthode. Pour réaliser une analyse interclasse, l'outil consulte le graphe d'appel complet afin de connaître la prochaine fonction à exécuter et son contexte d'exécution.

Le code de la méthode est parcouru instruction par instruction, et les éléments de la frame sont mis à jour en fonction de l'instruction visitée. Dans le cas où l'instruction visitée est de type

`invokeinstruction` (invocation d'une méthode), alors une nouvelle instance de `opcodeStack` est créée pour la nouvelle méthode visitée sans supprimer l'instance précédente (frame de la méthode qui l'invoque). Le code de la nouvelle méthode est parcouru et sa frame est mis à jour pour chaque instruction visitée. Quand l'analyse de la méthode invoquée est terminée, la frame est supprimée et l'analyse reprend au contexte de la méthode appelante pour continuer l'interprétation des instructions suivantes. Comme expliqué précédemment, nous tenons compte dans notre analyse de la dépendance causale inter-méthodes et nous propageons l'étiquetage aux valeurs de retour et aux paramètres des méthodes.

La propagation des étiquettes à l'intérieur de la méthode est réalisée dans la classe `opcodeStack` modifiée. Un fichier de journalisation est mis à jour à chaque détection d'une nouvelle vulnérabilité XSS. À la fin de l'analyse ce fichier contiendra l'ensemble des vulnérabilités détectées, indiquées par le numéro de ligne du code source où la faille a été détectée et le chemin d'exécution qui à conduit à cette faille.

Nous avons également implémenté l'annulation de l'étiquette pour des items filtrés. Sachant qu'il n'existe pas d'API de filtrage XSS compatible avec Java Card 3, nous avons développé l'API `JCXSSFilter` composée de cinq fonctions de filtrage, chacune correspondant à un point d'insertion particulier. L'étiquette d'un item est annulée s'il est filtré par une de ces fonctions. Cependant notre outil d'analyse statique ne permet pas de vérifier si la fonction utilisée est celle qui convient pour le point d'insertion courant.

6.3 Implémentation de *JCXSSFilter*

Pour utiliser l'API *JCXSSFilter*, le développeur doit l'importer et ensuite appeler ses fonctions pour filtrer une donnée. Elle est composée de cinq fonctions de filtrage.

1. `EncodeForHTMLEnteties()` : filtrage des données insérées dans le corps HTML, entre des balises HTML simples ;
2. `EncodeForHTMLattribut()` : filtrage des données non fiables injectées dans un attribut HTML ;
3. `EncodeForjavascript()` : encodage des données non fiables injectées dans un code JavaScript prévu dans l'application ;
4. `EncodeForCSS()` : filtrage des données non fiables injectées dans des attributs de style CSS ;
5. `EncodeforURL()` : filtrage des données non fiables injectées dans des attributs d'URL.

En plus de ses méthodes, nous avons implémenté la méthode `Validator()` qui est utilisée pour valider le format d'une URL.

6.4 Expérimentations

Ce chapitre décrit les résultats expérimentaux de notre outil d'analyse statique et de l'API de filtrage XSS présentée dans le chapitre précédent. L'analyse statique a été appliquée à deux catégories d'applications de référence. La première comporte un ensemble d'applications réelles conçues pour fournir des services bien particuliers et la deuxième est un large ensemble de programmes de base qui traitent une variété de cas qui peuvent se présenter dans une application concrète. Notre

objectif est de vérifier si notre outil d'analyse traite le maximum de cas possibles et de déterminer sa performance en temps d'exécution et le nombre de faux positifs générés. Nous avons également testé notre API de filtrage *JCXSSFilter* sur un prototype de cartes à puce Java Card 3, afin de tester son influence sur la performance en temps d'exécution.

6.4.1 Stratégie d'évaluation de l'outil d'analyse statique

Afin de réaliser un test pertinent de notre outil d'analyse statique, il est nécessaire de l'appliquer à des applications industrielles destinées à être utilisées sur le marché. Cependant, au jour où nous écrivons cette thèse, la carte à serveur Web embarqué en particulier la plateforme Java Card 3 n'a pas connu un grand succès. À notre connaissance, il n'existe aucune application Web pour carte à puce sur le marché. Néanmoins, nous avons exploité l'application « Disque virtuel » développée au sein de notre équipe (SSD) et présentée dans le chapitre 1, et une autre application nommée « JC3CreditDebit » qui est réalisée au centre CNAM de Paris.

Rappelons que l'application « Disque virtuel » a pour but de sécuriser l'accès à distance à des fichiers. Elle garantit que seul le propriétaire d'une carte à puce (confidentialité) contenant cette application, a le droit d'accès, en lecture ou en écriture, à des fichier chiffrés, sauvegardés sur des serveurs FTP distants. Les clés de chiffrement/déchiffrement sont sauvegardées dans la carte à puce.

L'application « JC3CreditDebit » développée par Thomas SOUVIGNET (étudiant du cnam de Paris) est une application de gestion de solde, utilisant deux types d'opérateurs : *Visa* et *MasterCard* au choix. Elle est composée de deux servlets de paiement qui partagent une interface SIO et qui gèrent les connexions externes.

Nous avons également développé un paquetage d'applications de test nommé *TestPackage* qui a été inspiré de l'application *Securibench Micro*[16] . Il est composé d'une série de programmes simples qui traitent différents cas possibles qui peuvent se retrouver dans une application concrète. Ces programmes sont classés en quatre catégories selon la propriété que nous voudrions tester. Les quatre sous paquetages de l'application *TestPackage* sont présentés ci-dessous. Chacun de ces paquetages est composé d'un ensemble de programmes :

- **basique** : ensemble de programmes traitant différents cas de manipulation de données de type `String et StringBuffer`.
- **conteneur** : ensemble de programmes traitant différents cas de manipulation des éléments d'un tableau ou d'un vecteur.
- **inter-procedurale** : ensemble de programmes traitant des cas de propagation des données entre des méthodes d'une même classe ou de classes différentes.
- **persistance** : ensemble de programmes traitant les cas d'objets persistants.

Le programme de test *listing 6.4* traite un cas de manipulation de variables de type `String`. L'ensemble des vulnérabilités sont indiquées par des commentaires : `/*Tainted*/` qui correspond à la source et `/*Bad*/` qui correspond à un état puits atteint à partir d'un état source (vulnérabilité). Ces commentaires nous facilitent la comparaison des résultats de notre analyse aux résultats attendus. Dans cet exemple la ligne 3 correspond à un état S, une étiquette est donc associée à la variable `s1`. Cette étiquette est propagée aux variables `s2`, `s3`, `s4`, `s5` et `s6` car elles sont le résultat

16. suite de programmes de test pour des analyseurs d'applications JEE http ://-suif.stanford.edu/ livshits/work/securibench-micro/

de la manipulation d'une variable étiquetée respectivement par les méthodes : `toUpperCase()`, `concat()`,`replace()`, `append()` et `substring()`(états D). À la ligne 12, la variable étiquetée `s6` est passée en paramètre de la méthode `HttpServletResponse.getWriter().println()`, l'exécution passe donc à un état final F et une vulnérabilité doit être signalée à ce niveau.

Listing 6.4– Exemple de programme de test contenu dans l'application *testPackage*

```
1  public class Basic6 extends BasicTestCase implements MicroTestCase
       {
2      protected void doGet(HttpServletRequest req,
           HttpServletResponse resp) throws IOException {
3          String s1 = req.getParameter("name"); /*Tainted*/
4          String s2 = s1.toUpperCase();
5          String s3 = s2.concat(";");
6          String s4 = s3.replace(';', '.');
7          String s5 = ":" + s4 + ":";
8          String s6 = s5.substring(s5.length() - 1);
9
10         PrintWriter writer = resp.getWriter();
11
12         writer.println(s6);    /* BAD */
13     }
```

6.4.2 Temps de l'analyse

Pour les deux applications « Disque virtuel » et « JC3CreditDebit », nous avons calculé le temps d'exécution en trois phases : la première correspond à *Findbugs* la deuxième à la génération du graphe d'appel interclasses et la dernière correspond à l'analyse par dépendance causale, qui inclut la détection des états S et F et la propagation de l'étiquetage.

Pour les deux applications testées nous avons constaté que la plus grande partie du temps a été occupée par le calcul du graphe d'appel complet ; cela s'explique par le fait que le graphe d'appels complet nécessite de parcourir toutes les classes de l'application testée et toutes les méthodes contenues dans chaque classe pour générer l'ensemble des flux possibles de l'application. *Findbugs* occupe également une grande partie du temps de l'analyse. Durant cette phase, *Findbugs* charge notre plugin *XSSDetector* (et d'autre plugins s'ils existent) et l'application à tester. Il effectue une première passe du bytecode pour lire les fichiers `class` et générer des objets `JavaClass`. Il lance ensuite une deuxième passe où les plugins sont exécutés. L'analyse par dépendance causale occupe la plus petite partie du temps. Cela est dû au fait que l'analyse parcourt uniquement les chemins d'exécution pré-calculés débutant à partir des méthodes qui traitent les requêtes HTTP. Le temps total de l'analyse est faible dans les deux cas d'applications testées.

Application	taille	**Findbugs**	**construction du CFG interclasses**	**analyse**
JC3CreditDebit	68 ko	843 ms	710 ms	359 ms
Disque Virtuel	100 ko	842 ms	825 ms	452 ms

Table 6.1 – Temps de l'analyse statique

6.4.3 Détection de vulnérabilités

L'utilisation de notre outil pour vérifier les applications « Disque virtuel » et « JC3CreditDebit» a permis de révéler des vulnérabilités XSS persistantes et volatiles dans ces applications. Les résultats sont présentés dans le tableau 6.2.

Application	**XSS volatiles**	**XSS persistantes**	**total**	**faux positifs**
JC3CreditDebit	8	2	10	3
Disque Virtuel	5	4	9	1

Table 6.2 – Résultat d'analyse des applications « Disque virtuel » et « JC3CreditDebit »

Dans l'application « JC3CreditDebit », l'utilisateur est invité à entrer un montant à débiter ou à créditer. Le solde est ensuite sauvegardé de manière persistante dans la carte et il est affiché à plusieurs niveaux dans les pages Web de l'application. Vu que les valeurs entrées par l'utilisateur ne sont pas filtrées, alors des vulnérabilités XSS volatiles et persistantes sont détectées par notre outil d'analyse. Le code du *listing 6.5* est un cas où un faux positif a été généré.

Listing 6.5– Faux positif généré dans l'analyse de JC3CreditDebit

```
1   public void doGet(HttpServletRequest request,
        HttpServletResponse response) throws IOException,
        ServletException {
2     //Creation du lien avec la SIO
3     PMSIO pm = (PMSIO) ServiceRegistry.getServiceRegistry().
          lookup(SIO_URI);
4
5     ...
6     //Creation d'un canal de reponse
7     PrintWriter out = response.getWriter();
8     String login = request.getParameter("login");
9     String pass = request.getParameter("password");
10    if(!pm.isLoginCorrect(login, pass)) {
11        //Si mot de passe incorrect
12        setHeader(response, out, "Vous etes deconnectes");
13        out.println("Vous etes deconnectes<br>");
14
```

```
15                //Fermeture de la page
16                setFooter(response, out);
17            }
18            else {
19                ...
20                //login et password retournes en sortie
21                out.println("<td colspan=2><input type=\"hidden\" name
                      =\"login\" value=\""+login+"\"><input type=\"hidden
                      \"          name=\"password\" value=\""+pass+"\">);
22            }
23    }
```

Dans ce code les données d'authentification (login et le mot de passe) d'un utilisateur authentifié sont affichées dans l'URl d'une page Web générée par l'application (ligne 21). Bien que l'affichage des données confidentielles ne soit pas une bonne pratique (un attaquant peut les capturer et les utiliser pour usurper l'identité de l'utilisateur légitime), aucune vulnérabilité XSS n'est présente dans ce cas. En effet, le login et le mot de passe sont comparés à une liste d'identifiants valides par la méthode `isLoginCorrect(login, pass)` (ligne 10) et l'URL n'est affichée que dans le cas où ces identifiants sont valides (ligne 18). Vu que notre analyse ne vérifie pas la véracité de la condition dans une instruction de branchement conditionnelle, alors une vulnérabilité est signalée générant un faux positif.

L'application « Disque virtuel » est composée de plusieurs points d'entrée où l'utilisateur est invité à ajouter un fichier ou serveur par son nom et son adresse. Ces valeurs sont sauvegardées de manière persistante dans l'application et sont retournées dans plusieurs sorties, générant ainsi des vulnérabilités XSS volatiles et persistantes. Les vulnérabilités détectées sont dues à la sauvegarde de ces données de manière persistante dans l'application, et leur utilisation dynamique dans certaines pages Web de l'application qui fournit la liste des noms de serveurs sauvegardés dans l'application ou la liste des fichiers disponibles dans un serveur.

Notre outil a détecté des vulnérabilités XSS (persistantes) même dans le cas où un objet dont un attribut étiqueté, est sauvegardé dans un vecteur persistant.

Les résultats des tests que nous avons réalisés sur notre application de référence *TestPackage* sont présentés dans le tableau 6.3. La deuxième colonne du tableau nous donne le nombre de vulnérabilités prévues pour chacune des propriétés testées, la troisième colonne présente les résultats obtenus par notre analyse, et la quatrième colonne nous informe sur le nombre de faux positifs obtenus.

Catégorie de test	**Vulnérabilités implémentées**	**vulnérabilités détectées**	**faux positifs**
basique	37	40	3
conteneur	14	18	4
inter-procédural	14	17	3
persistance	20	24	4
Total	**85**	**99**	**14**

Table 6.3 – Résultats d'analyse de l'application *TestPackage*

Les résultats que nous avons obtenus sont satisfaisants. Tous les cas de vulnérabilité que nous avons prévus dans notre application de test ont été détectés (analyse complète). Cependant, quelques faux positifs ont été générés, mais leur taux n'est que de 14%. Les faux positifs sont une caractéristique de tous les outils d'analyse statique. Ils sont généralement générés en présence de conteneurs ou des instructions de branchement conditionnels.

Dans le cas des conteneurs (tableau, vecteur), notre outil ne se limite pas à l'étiquetage de l'objet conteneur mais il applique un étiquetage des éléments du tableau aussi. En effet, si nous nous limitons à étiqueter le conteneur alors tous ses éléments seront considérés comme non fiables, même si seulement un élément l'est réellement. Ainsi, notre implémentation nous réduit considérablement le nombre de faux positifs.

Les faux positifs. Une partie des faux positifs générés par notre outil est due au fait que dans certains cas des données étiquetées qui sont filtrées sont considèrées comme non fiables. L'exemple du *listing 6.6* illustre un cas où il serait difficile de savoir si la donnée est fiable ou pas.

Dans cet exemple, nous définissons une classe `element` contenant un attribut `a1` et un vecteur `tab` persistant (défini par le modificateur `static`) d'objets instances de la classe `element`. Notre outil d'analyse statique détecte à la ligne 3 un état S, il associe donc une étiquette à la variable `s1`. Cette étiquette est également associée à l'objet `e1` auquel appartient cette donnée.

À la ligne 8, la variable s1 est filtrée en utilisant notre API de filtrage *JCXSSFilter*. L'étiquette qui lui est associée est donc retirée. Cependant, l'objet `e1` reste étiqueté. En effet, il serait risqué de retirer l'étiquette à cet objet car il est possible que d'autres attributs le composant soient non fiables. Ainsi, à la ligne 10, notre outil génère un faux positif car il considère qu'un objet (`e1`) étiqueté est sauvegardé dans un vecteur persistant.

Listing 6.6– Programme de test sur les vecteurs

```
1  public class vecteur03 extends BasicTestCase implements
       MicroTestCase {
2      protected void doGet(HttpServletRequest req,
           HttpServletResponse resp) throws IOException {
3          String s1 = req.getParameter("name");       /*Tainted*/
4          static Vector tab=new Vector <element>();
5          element e1=new element(s1);
6          element e1=new element("toto");
7
8          JCXXFilter.encodeforHTML(e1.get());
9
10         tab.addelement(e1);                /*OK*/
11         tab.addelement(e2);                /*OK*/
12       }
13         ...
14         public class element{
15             private String a1;
16             public element(String str){
```

```
17                  a1=str;
18              }
19
20          public String get(String str){
21                  a1=str;
22              }
23      }
24  }
```

6.4.4 Faiblesse de l'analyse statique

Une analyse statique performante se doit de ne pas générer de faux négatifs et présenter le moins possible de faux positifs. Cependant nous avons détecté quelques cas particuliers qui génèrent de fausses alertes et qu'il serait difficile, de traiter par une analyse statique :

Branchements conditionnels

Dans certains cas, les faux positifs sont inévitables pour une analyse statique, en particulier en présence d'instructions de branchement conditionnel. L'exemple 6.7 illustre un de ces cas.

Dans le code suivant, les lignes 5 et 10 correspondent à des branchements conditionnels utilisant l'instruction `if`. Les deux conditions correspondantes ne peuvent pas être toutes les deux à la fois vérifiées (*true*). En effet, un entier `i` ne peut pas être à la fois supérieure à 1 et inférieur à 2. Quand la première condition est vérifiée, l'instruction de la ligne 6 est exécutée, celle-ci présente un état S ; la variable `X` doit donc être étiquetée. Quand la deuxième condition est vérifiée l'instruction de la ligne 11 est exécutée, celle-ci est une invocation de la méthode `HttpServletResponse.getWriter().println()` qui si ses paramètres sont étiquetés, alors une vulnérabilité est détectée. Mais comme la deuxième condition ne peut pas être vérifiée, si la première l'est alors, ce code ne présente aucune vulnérabilité.

Cependant, notre outil ne traite pas la véracité de la condition dans une instruction de branchement conditionnel mais elle considère les deux cas possibles (vrai et faux). Une vulnérabilité est donc signalée pour cet exemple, générant un faux positif. Une amélioration consiste à vérifier l'accessibilité de chaque condition par résolution des contraintes.

Listing 6.7– Faux positif indétectable par une analyse statique

```
1  class branchement05{
2
3      String X="toto";
4
5      if(i>1){
6          X=getParameter(); /*Tainted*/
7      }else
8          X="toto";
9
```

```
10     if(i<2)
11         out.println(X); /*OK*/
12 }
```

Ordre d'exécution des méthodes HTTP

Notre outil analyse une application en parcourant son graphe d'appel à partir d'une méthode de la classe `HTTPServlet` qui traite les requêtes HTTP (doGet, doPost, etc). L'analyse consiste à chercher ces méthodes (points d'entrées) dans le graphe d'appel complet pré-construit et à parcourir à partir de ces points d'entrées tous les flux qui en découlent. Cependant, une application peut contenir plusieurs méthodes de la classe `HTTPServlet`, et l'ordre de vérification de ces différentes méthodes peut influencer sur les résultats de l'analyse.

En effet, supposons le cas où une page Web est composée de deux liens, l'un correspond à une requête GET traitée par une méthode doGet() d'une servlet SERVLET et l'autre est une requête POST traitée par la méthode doPost() de la même servlet. Supposons aussi une variable `X` (variable d'instance) qui est partagée par ces deux méthodes *(Listing 6.8)*.

À la ligne 6 du *listing 6.8* notre analyse statique va étiqueter la variable `X` car la valeur de retour de la méthode `request.getParameter()` lui est affectée. La ligne 10 correspond à l'invocation de la méthode `HttpServletResponse.getWriter().println()` avec la variable `X` en paramètre, cela correspond à une potentielle vulnérabilité XSS. Cette vulnérabilité n'est effective que dans le cas où la méthode doGet() est analysée avant la méthode doPost(). Ainsi l'analyse statique peut donc engendrer des erreurs (faux positifs ou faux négatifs) suivant son ordre d'analyse. En effet, si notre outil commence par l'analyse à la méthode doPost(), la potentielle vulnérabilité ne sera pas détectée.

Pour palier à ce problème nous proposons de traiter les différents cas possibles d'ordre d'exécution des méthodes HTTP. Une autre solution moins exhaustive est présentée dans les perspectives.

Listing 6.8– Ordre d'exécution des méthodes HTTP

```
1  public class inter-procedural10 extends BasicTestCase implements
       MicroTestCase {
2      static String X;
3
4      protected void doGet(HttpServletRequest req,
           HttpServletResponse resp) throws IOException {
5
6          X=request.getParameter();
7      }
8
9      protected void doPost(HttpServletRequest req,
           HttpServletResponse resp) throws IOException {
10         PrintWriter out = resp.getWriter();
11         out.println(X);
```

```
12      }
13  }
```

Appels interclasses via des scripts

Dans le cas précédent, l'ordre d'exécution des méthodes est aléatoire et ne peut être connu au préalable car il dépend du comportement de l'utilisateur en interagissant avec l'application. Dans cette partie, nous présentons un autre cas particulier où l'ordre d'exécution peut être spécifié dans l'application via des scripts (JavaScript) mais que notre analyse n'est pas capable de détecter. Afin de mieux comprendre, supposons un exemple d'application composée de deux Servlets : *Accueil*, *Servlet1*.

La servlet *Accueil* fournit une page Web constituée d'un bouton dont un clic exécute la méthode du script `fonction.js`. La fonction `envoi()` contient un appel à la servlet SERVLET par son URI. La correspondance entre l'URI et le nom de la servlet est précisée dans le fichier de configuration *web.xml*.

Listing 6.9– Appel interclasses via un code JavaScript

```
1   public class Accueil implement HTTPServlet{
2       public static String X="toto";
3
4       protected void doGet(HttpServletRequest req,
            HttpServletResponse resp) throws IOException {
5           ...
6
7           X=req.getParameter("name");
8           ...
9           out.println("<script language=\"javascript\" type=\"text/
                javascript\" src=\"fonctions.js\"></script>");
10
11          out.println("<td ><input type=\"button\" name=\"envoyer\"
                value=\"envoyer\"onClick=\"envoi()\"></td>");
12          ...
13
14   }
15   public class Servlet1 implement HTTPServlet{
16
17
18          protected void doGet(HttpServletRequest req,
                HttpServletResponse resp) throws IOException {
19          ...
20          PrintWriter out = resp.getWriter();
21          ...
22          out.println(Accueil.X);
```

```
23          ...
24
25    }
```

Pour construire le graphe d'appel complet, notre outil d'analyse parcourt tous les fichiers `class` de l'application. Cependant la méthode `doGet()` de la classe *Accueil* appelle la servlet *servlet1* via un script. Pour pouvoir détecter cet appel l'analyse statique devrait analyser successivement :

1. le code HTML en paramètre de la méthode `println()` dans la servlet *Accueil* afin de détecter l'appel au script (ligne 11),
2. le script `fonctions.js` pour détecter l'appel à la servlet *servlet1* via son URI, et enfin
3. le fichier de configuration Web.xml pour détecter la servlet correspondant à l'URI.

Cette analyse serait compliquée à réaliser et nécessiterait en plus de parcourir du bytecode, d'analyser du code HTML et du code JavaSript. Notre outil se limite à analyser des fichiers `class`. Il n'est donc pas en mesure de propager l'étiquetage entre une méthode appelante et une méthode appelée via un autre moyen qu'une invocation Java. Dans le cas de l'exemple 6.9, si notre outil commence son analyse par la méthode `doGet()` de `servlet1`. La vulnérabilité XSS ne sera pas détectée (faux négatif).

6.4.5 Surcoût du filtrage

L'API de filtrage que nous avons développée est une archive au format .jar qui doit être chargée en mémoire persistante non modifiable : la ROM. Pour être utilisée dans une application, le développeur doit l'importer. Elle est composée de cinq fonctions que le développeur doit utiliser avec prudence en choisissant la fonction qui convient au point d'insertion qu'il voudrait sécuriser. La taille de l'API *JCXSSFilter* est de 30ko. Par conséquent elle est largement supportée dans la ROM d'une Java Card 3 dont la taille est supérieure à 256 Ko. Cette API serait également supportée en terme de taille dans une carte à puce Java Card 2.2 SCWS dont la taille de la ROM est supérieur à 64 ko, cependant son implémentation est basée sur le langage Java Card 3 (utilisation des types : `Vector, String, StingBuffer`). Des modifications seraient donc nécessaires pour porter cette API dans des cartes SCWS.

Pour tester l'influence de l'utilisation de *JCXSSFilter* sur la performance en temps d'exécution, nous l'avons appliquée dans l'application « Disque virtuel » . Nous avons utilisé dans un premier temps l'application « Disque virtuel » pour calculer le temps d'exécution des requêtes possibles et nous l'avons comparé au temps obtenu en utilisant les fonctions de *JCXSSFilter*. La comparaison a montré que la différence était inconsidérable (de l'ordre de quelque nanosecondes). Cependant, notons que vu les contraintes de ressources de la carte à puce, les applications qui lui sont dédiées sont généralement de petite taille comparées à des applications standards. Les points d'insertion seraient donc moins importants qu'une application Web classique. En conséquence le nombre d'appels aux méthodes de notre API ne peut pas être important. Des résultats expérimentaux que nous avons obtenus, nous déduisons que *JCXSSFilter* apporte de la sécurité dans les cartes à puce sans influencer leurs performance.

6.5 Conclusion

Dans ce chapitre, nous avons expliqué la façon dont nous avons implémenté les différents points essentiels de notre analyse. Nous avons également présenté les résultats expérimentaux sur des applications concrètes et une application de test. Les résultats que nous avons obtenus ont montré l'efficacité de notre outil qui génère un taux minime de faux positifs. Nous avons également testé notre API de filtrage *JCXSSFilter* pour vérifier sa portabilité sur la carte et son influence sur le temps d'exécution. Les résultats obtenus montrent que notre API *JCXSSFilter* est parfaitement supportée par des plateformes Java Card 3. Cependant de potentielles améliorations peuvent être apportées, telles que l'analyse de propagation des contraintes qui permet la prédiction des branchements, et une analyse du code HTML pour vérifier si une fonction de filtrage correspond au contexte dans lequel elle est appelée.

Troisième partie

Sécurité du protocole HTTP

Chapitre 7

Le protocole HTTP et les techniques de vérification

Sommaire

7.1 Introduction **105**
7.2 Les techniques de vérification et de validation de protocoles **106**
7.2.1 La vérification formelle 106
7.2.2 Test de logiciels 107
7.3 Le Fuzzing **108**
7.3.1 Définition 108
7.3.2 État de l'art des logiciels de *fuzzing* 109
7.3.3 Le *fuzzing* sur carte à puce 110
7.4 Le protocole HTTP **111**
7.4.1 Historique 111
7.4.2 Principe de fonctionnement 111
7.4.3 Les requêtes HTTP 111
7.4.4 Les réponses HTTP 113
7.4.5 Les champs d'entête 114
7.4.6 Accès aux ressources protégées 115
7.4.7 Antémémoire 115
7.5 Conclusion **116**

7.1 Introduction

La multitude et la sensibilité des domaines d'utilisation de la carte à puce ont multiplié les recherches sur les mécanismes de sécurisation au niveau matériel. Cependant, la sécurité de ces plateformes est également liée aux vulnérabilités au niveau applicatif. L'utilisation d'un serveur Web expose la carte à puce à des menaces supplémentaires. Pour ces raisons, il est nécessaire de garantir qu'aucune vulnérabilité au niveau des protocoles installés sur la plateforme n'est présente dans la carte. Il existe deux façons d'éliminer les vulnérabilités :

– l'utilisation des méthodes formelles dans le développement de logiciels sans faille, conformes à leur spécification.
– la génération d'une suite de tests afin de vérifier que certaines propriétés de sécurité sont respectées dans la mise en œuvre de l'application.

Souvent les modèles formels vérifient uniquement si le logiciel est conforme à sa spécification, tandis que l'approche de test, en particulier le test par *fuzzing*, permet de détecter des failles de sécurité dans l'application mise en œuvre. La carte à puce est un dispositif très limité et le protocole HTTP est un protocole complexe, sujet aux erreurs. Notre travail a pour but de tester la conformité et la robustesse de l'implémentation du protocole HTTP dans les serveurs Web, en particulier, les serveurs Web embarqués. Nous avons choisi la technique de *fuzzing* pour son efficacité dans la vérification des applications et systèmes.

Ce chapitre est constitué principalement de deux parties. La première est un état de l'art des techniques de vérification de protocoles, en particulier le *fuzzing*. La deuxième partie présente le protocole HTTP et ses principales caractéristiques.

7.2 Les techniques de vérification et de validation de protocoles

La complexité dans la conception d'un protocole réside dans la gestion de toutes les actions et réactions possibles. Cela signifie qu'il faut prendre en compte toutes les commandes et les séquences de commandes possibles. La vérification d'une application consiste à vérifier qu'elle fonctionne correctement, respectant des propriétés prédéfinies ; les techniques de vérification de protocoles sont généralement basées sur les méthodes formelles. La validation d'une application consiste à la tester dans le but de révéler la présence d'erreurs ou des cas de non-conformité à la spécification. Nous détaillerons ces deux techniques dans ce qui suit.

7.2.1 La vérification formelle

Pour vérifier qu'une spécification respecte tous les besoins et que le protocole implémenté respecte sa spécification, diverses approches ont été utilisées. Une catégorie d'outils de vérification de protocoles est basée sur les techniques de spécification formelle dont les modèles de transition [Boc03, Hol91], les réseaux de Petri [YX11], les langages de modélisation formelle [MF05] et la logique temporelle [CMcMW04].

Les modèles de transitions [STE+82] sont les plus souvent utilisés dans la modélisation des protocoles de communication car ces derniers sont basés sur un ensemble d'événements tels que des commandes ou des événements de synchronisation. Les machines à états finis modélisent un protocole dont les événements forment ses entrées. D'autres méthodes qui ne nécessitent pas de définition d'état explicite peuvent également être utilisées. Une variante des systèmes à transitions consiste à définir les séquences d'entrées et sorties autorisées et leurs relations.

Après la modélisation du protocole, il est nécessaire de vérifier la validité de son implémentation. La validation de protocoles, consiste à vérifier l'absence d'erreurs logiques de spécification telles que les erreurs d'inter-blocages, les réceptions non spécifiées, etc. La spécification algébrique [BMV05] est une technique largement utilisée dans la validation de protocoles cryptographiques [CDL06]. Elle

consiste à vérifier le protocole en tenant compte de toutes les propriétés algébriques de l'algorithme de chiffrement utilisé. L'analyse d'accessibilité standard est généralement l'approche la plus utilisée. Elle se base sur une modélisation par automate à états finis. L'état global du système est exprimé pour chaque transition. Chaque état global est ensuite analysé. Une recherche exhaustive vérifie que tous les états sont accessibles en toute sécurité.

L'analyse d'accessibilité est appropriée pour déterminer si un protocole est correct par rapport à sa spécification, mais elle ne garantit pas qu'il soit sécurisé. Une autre limitation importante de cette technique est l'obligation de prendre des hypothèses radicales afin de garder un espace d'états réduit.

7.2.2 Test de logiciels

La validation d'un logiciel en utilisant la méthode basée sur des tests consiste à exécuter le logiciel en lui appliquant un jeu de tests dans l'intention de détecter des anomalies ou des défauts (bogues) ou de révéler la présence d'erreurs de non-conformité à la spécification. Cette approche se base généralement sur la comparaison des résultats d'exécution des tests sur l'application aux résultats attendus [BGLP08, UL07].

La norme IEEE *(Standard Glossary of Software Engineering Terminology)* définit le test par :

« Le test est l'exécution ou l'évaluation d'un système ou d'un composant, par des moyens automatiques ou manuels, pour vérifier qu'il répond à ses spécifications ou identifier les différences entre les résultats attendus et les résultats obtenus. »

Les outils de test peuvent être classés selon : le moyen d'accessibilité, la stratégie d'évaluation ou le critère d'évaluation.

1. **Classification par accessibilité :** le système ciblé peut être fourni avec ou sans son code source. Nous distinguons :
   - *le test en boite noire :* cette catégorie suppose n'avoir aucune connaissance sur l'implémentation de la cible. Les tests sont sélectionnés à partir d'une spécification du système ciblé (formelle ou informelle). Le *fuzzing* est la méthode la plus utilisée dans ce cas.
   - *le test en boite blanche :* consiste à sélectionner des tests à partir de l'analyse du code source du système.
2. **Classification par stratégie :** dans le cycle de vie d'un logiciel trois types de test peuvent intervenir :
   - *les tests unitaires* : test des parties d'un programme indépendamment les unes des autres ;
   - *le test d'intégrité* : test de la composition des modules via leur interface (vérification des communications entre modules, appels de procédures, etc.) ;
   - *le test système* : test de la conformité du produit fini par rapport ce qui est prévu dans le cahier des charges. Ces tests sont réalisés en boîte noire via l'interface du système.
3. **Classification par critère d'évaluation :** les méthodes de test peuvent être différentes selon le critère que nous souhaitons évaluer :
   - *conformité* : consiste à générer des tests à partir de la spécification et de vérifier que toutes les fonctionnalités prévues sont implantées selon leur spécification ;
   - *robustesse* : contrairement au test de conformité, les tests générés dans ce cas ne respectent pas les comportements spécifiés dans le but de détecter si le système gère les utilisations imprévues.

- *performance* : consiste à appliquer les tests à différents niveaux de charge d'utilisateurs et de mesurer la performance du système (temps de réponse du système, l'utilisation des ressources, etc.)
- *sécurité* : consiste à appliquer une suite d'attaques sur la cible pour découvrir les vulnérabilités et les faiblesses du système.

Les méthodes de spécification formelle se basent sur un modèle de protocole et une procédure pour vérifier des propriétés bien définies. Leur inconvénient est principalement le temps nécessaire à la conception du modèle et la nécessité de connaître l'implémentation de l'application vérifiée. Les approches basées sur le modèle de transitions sont très vite limitées par l'explosion combinatoire des états générés. D'autre part, la validation par test permet de détecter des failles de sécurité et de vérifier la conformité et la robustesse des applications. Notre objectif étant de valider l'implémentation du protocole HTTP dans les cartes puce (en boite noir), nous avons choisi la technique de test par *fuzzing* que nous présentons dans la section suivante.

7.3 Le Fuzzing

7.3.1 Définition

Le *fuzzing* est une technique de détection d'erreurs d'implémentation logicielle par injection de données invalides. Le but principal du *fuzzing* est de réussir à faire passer le système, l'application ou le protocole dans un état non souhaité ou inattendu. On distingue trois techniques de génération de données de *fuzzing* [TDM08] :

- **la génération aléatoire** : cette technique présente l'inconvénient d'être "aveugle", générant un nombre important de données de test dont une grande partie pourrait être rejetée par le système ciblé. En effet, celui-ci rejettera les données dès qu'elles ne correspondent plus à ce qui est attendu ;
- **les modèles de données** : il s'agit d'une technique de génération semi-aléatoire. Elle se base sur un format descriptif des données valides que les données de *fuzzing* doivent respecter. L'efficacité de cette technique n'est plus à prouver, à condition que la spécification soit bien modélisée. Cependant l'écriture des modèles de données est un processus complexe et fastidieux et nécessite de disposer de la spécification du système ou de l'application ciblée ;
- **Les mutations** : c'est une alternative entre les deux précédentes techniques. Elle consiste à muter une session connue et valide (fichier, réseaux, etc). Contrairement à la technique précédente, celle-ci ne requièrt pas de connaissance particulière du protocole car elle utilise seulement une session connue et la transforme. Elle est donc facilement mise en œuvre mais ne traite que les cas présents dans le fichier de test.

Il existe également des outils qui appliquent une combinaison de ces différentes techniques [MP07, Micb].

La figure 7.1 représente un modèle général d'un outil de *fuzzing*. Tous les outils de *fuzzing* partagent un ensemble de fonctions semblables, à savoir :
- la génération des données de test suivant une logique propre à l'outil,
- la transmission des données vers la cible,
- la surveillance et la sauvegarde des réponses et réactions de la cible,

– et éventuellement l'automatisation de l'analyse des résultats (réduire autant que possible l'intervention de l'utilisateur dans l'analyse).

Les deux derniers éléments pourraient être considérés comme facultatifs ou être mis en œuvre en dehors de l'outil de *fuzzing*.

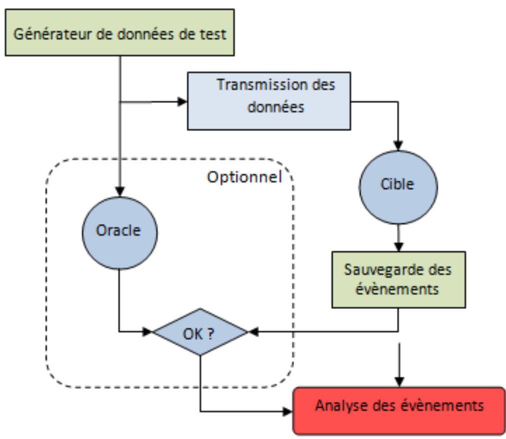

Fig. 7.1 – Principe du *fuzzing*

7.3.2 État de l'art des logiciels de *fuzzing*

Les outils de test par *fuzzing* peuvent être répartis en deux types. Les outils spécifiques permettent de tester un programme ou un protocole particulier, alors que les logiciels de *fuzzing* proposent une interface de programmation permettant l'implémentation d'outils de test à différents niveaux (fichiers, API, argument de lignes de commandes, etc). Dans ce qui suit nous présentons quelques exemples de logiciels, *open source*, les plus connus :

– **SPIKE :** ce logiciel [Ait04] est écrit en langage C. Il est dédié en particulier à développer des outils de *fuzzing* pour des protocoles réseaux. Il a été à l'origine du concept de blocs qui a été repris dans plusieurs outils par la suite, dont Peach et Sulley, etc. Les structures de données sont réparties et représentées sous forme de blocs. Cette approche permet de faire abstraction de la construction des différentes couches d'un protocole ;
– **Peach :** est une plateforme de *fuzzing* [Micb] développée en Python et basée à la fois sur les modèles de données et les mutations, dans la génération des données de test. Elle applique également le concept de blocs définis dans SPIKE. L'utilisation de cette plateforme nécessite de définir un ensemble de fichiers XML, appelés *Pits*, qui décrivent le protocole ciblé et la configuration du processus de *fuzzing*. Chaque fichier peut être composé de plusieurs blocs et spécifier les champs sur lesquels le *fuzzing* sera appliqué.
Peach présente l'avantage d'être flexible, permettant à l'utilisateur de définir sa propre stratégie de *fuzzing* (modification d'une ou de plusieurs données, définition des mutants à utiliser, modification des parties du modèle de données, changement des flux des modèles d'états, etc). Un autre avantage de Peach est qu'il permet de configurer un *fuzzing* parallèle sur plusieurs cibles à la fois.
– **Sulley :** [P.A07] il se base également sur la technique de génération de données par blocs. Étant principalement conçu pour tester des protocoles réseau, la transmission des données

est implémentée par le mécanisme de sockets. Il utilise un ensemble d'agents permettant de gérer différentes cibles (communications réseau, détection d'éventuelles failles, contrôle de la machine virtuelle VMWare). Il offre plusieurs types élémentaires permettant de construire des trames (`static, bit-field, size`, etc).

- **Fusil :** [I.M] est une librairie développée en Python, utilisée pour écrire des programmes de *fuzzing*. Elle a été conçue à la base pour les programmes GNU/Linux en ligne de commande. Elle utilise des agents qui s'échangent des messages pour déclencher des actions. Fusil offre diverses manières de rendre aléatoire les données qui sont transmises à l'application. Il offre aussi des mécanismes pour injecter des données incorrectes via la ligne de commande, les variables d'environnement, les fichiers de données, ou le réseau.

7.3.3 Le *fuzzing* sur carte à puce

La plupart des outils de *fuzzing* sont dédiés à tester des protocoles réseau tels que les protocoles TCP/IP, mais leur utilisation ne se limite pas à cet environnement. En effet, cette technique a été utilisée dans d'autres domaines tels que les applications en lignes de commandes, les formats de fichiers ou les communications interprocessus.

Le *fuzzing* a également été appliqué dans le test des applications et protocoles pour cartes à puce. Les premiers travaux [joi09] consistaient à tester les applets par injection d'une suite de commandes APDU invalides dans une entrée de l'applet, dans le but de la forcer à sortir de son contexte d'exécution prévu (*buffer overflow*) et de détecter des failles qui sont exploitées par la suite pour réaliser des attaques.

Des travaux plus récents se sont intéressés au test des protocoles implémentés dans les cartes à puce. M.Barreaud [BLIC11, MICL11] dans ses travaux a démontré l'efficacité de cette technique dans le test de la robustesse et de la conformité du protocole BIP implémenté dans les cartes à puce à serveur Web embarqué, par-rapport à la spécification définie par ETSI. L'outil proposé est basé sur le logiciel de *fuzzing* Peach. Dans les travaux présentés par Lancia [Lan11], le logiciel Sulley est utilisé pour tester et détecter des failles dans un protocole de paiement sécurisé pour les cartes à puce EMV. Dan Griffin [Gri] propose un outil nommé SCFuzz, destiné à tester les *Middlewares* des cartes à puce. En générant des exceptions, SCFuzz va permettre de se servir des dysfonctionnements observés afin de découvrir éventuellement des vulnérabilités exploitables.

Nous avons utilisé la technique du *fuzzing* dans l'objectif de tester la robustesse et la conformité du protocole HTTP implémenté dans les cartes à puce à serveur Web embarqué. L'outil que nous avons développé est basé sur le logiciel **Peach**. Notre choix pour ce logiciel est principalement dû au fait qu'il a été éprouvé par plusieurs recherches, en particulier les travaux de M.Barreaud qui ciblent le même environnement que celui auquel nous nous intéressons (cartes à puces à serveur web embarqué). D'autre part, Peach permet de réaliser les tests sur plusieurs plateformes en parallèle, ce qui est très avantageux pour nous, vu les fortes contraintes en terme de ressources des cartes à puce. Dans la section suivante, nous présentons les principales caractéristiques du protocole HTTP nécessaires à la compréhension du fonctionnement de notre outil présenté dans le chapitre suivant.

7.4 Le protocole HTTP

HTTP (*Hyper Text Transfert Protocol*) [FGM+99] est le protocole de communication le plus utilisé sur le Web. Il permet d'acheminer un flux d'informations suivant l'architecture client/serveur. Le client est généralement le navigateur Web (Mozilla, Konqueror, etc.) qui envoie des requêtes à un serveur (Apache, IIS, etc.) contenant des données. Un paquet HTTP est le plus souvent encapsulé dans un paquet TCP qui lui-même est encapsulé dans des paquets IP et le tout dans des trames Ethernet.

Un serveur peut contenir diverses ressources de différents types notamment celles au format HTML. La localisation de ces ressources est basée sur le concept des URIs. C'est un moyen simple pour un humain de mémoriser l'emplacement d'une ressource sur internet et d'indiquer également le moyen d'y accéder. Les URIs se décomposent en deux catégories :

- URL (*localisateur uniforme de ressource*) qui indique l'emplacement dans sa forme absolue d'une ressource (ex : `http ://www.backtrack-linux.org/BT5-GNOME-ARM.7z`) ;
- URN (*nom uniforme de ressource*) : qui identifie les ressources et non pas son emplacement.

7.4.1 Historique

A l'heure d'aujourd'hui, le protocole HTTP a connu trois évolutions majeures. Le tableau 7.1 récapitule brièvement l'évolution du protocole en indiquant la principale caractéristique de chacune d'entre elles.

Version	**Date**	**RFC**	**Caractéristiques**
HTTP/0.9	1989	Non normalisée	Communication basique
HTTP/1.0	1996	1945	Support des formats MIME
HTTP/1.1	1999	2616	Connexions persistantes

Table 7.1 – Versions du protocole HTTP

7.4.2 Principe de fonctionnement

Le protocole HTTP repose sur une architecture client/serveur. Une application de ce genre suit la trame d'échange suivante (*figure 7.2*)

- le client transmet une requête vers le serveur, contenant des informations sur la ressource demandée ;
- le serveur renvoie la ressource demandée si disponible ou, le cas échéant, un message d'erreur.

7.4.3 Les requêtes HTTP

La requête transmise par le client au serveur est composée d'un ensemble de lignes (*listing 7.1*), dont :

1. **la ligne de requête** : contenant la méthode utilisée, l'URL de la ressource demandée et la version HTTP utilisée.

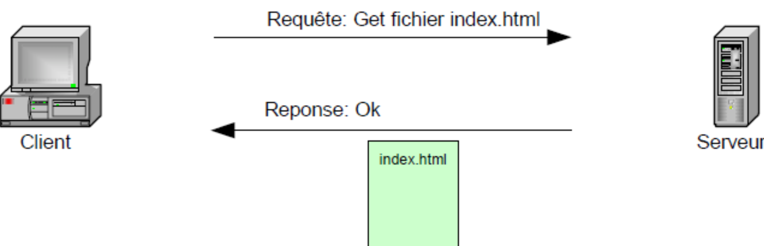

Fig. 7.2 – Communication en HTTP

2. **les champs d'entête de la requête** : il s'agit d'un ensemble de lignes facultatives permettant de donner des informations supplémentaires sur la requête et/ou le client (navigateur, système d'exploitation, ...). Chacune de ces lignes est composée d'un nom spécifiant le type d'entête et d'une valeur correspondante.
3. **le corps de la requête** : la requête peut optionnellement contenir un corps (des données) qui est notamment utilisé pour transmettre des paramètres avec la méthode POST. Le corps est transmis après les lignes d'entêtes, il est séparé du dernier entête par une ligne vide spécifiée par un double CRLF (*carriage return et linefeed*).

Listing 7.1– Exemple de requête HTTP

```
1 GET /index.html HTTP/1.1
2 Host: www.example.com
3 Accept: */*
4 Accept-Language: fr
5 User-Agent: Mozilla/4.0 (MSIE 6.0; windows NT 5.1)
6 Connection: Keep-Alive
```

Les méthodes HTTP

Le type des requêtes HTTP est spécifié par des méthodes qui définissent l'action demandée au serveur. La norme HTTP/1.1 définit 11 méthodes parmi lesquelles 8 sont le plus souvent implémentées. Nous les présentons ici :

- **OPTIONS** : permet de demander des informations sur les options de communication disponibles sur la chaine requête/réponse ;
- **GET** : est utilisée pour récupérer des informations spécifiées par une URI. Elle peut être conditionnelle en utilisant les champs « If-Modified-Since », « If-Match » ou partielle en utilisant le champ "Range" permettant de spécifier une partie de la ressource demandée ;
- **HEAD** : est identique à GET, à une exception près. Le serveur ne doit retourner que l'entête. Elle est utilisée pour obtenir des informations sur une ressource sans en transférer l'intégralité ;
- **POST** : est utilisée pour envoyer un nombre conséquent de données vers un script hébergé par le serveur qui va les traiter pour créer ou mettre à jour une ressource. Elle est plus souvent utilisée dans les formulaires HTML ;

- **PUT** : permet à un utilisateur de mémoriser un document sur le serveur (l'URI de demande). La différence entre la méthode PUT et POST est que l'URI dans une demande POST identifie la ressource qui va traiter l'entité. A l'inverse, l'URI dans la demande PUT identifie l'entité ;
- **DELETE** : demande au serveur la suppression de la ressource identifiée par l'URI. Elle réalise l'action inverse de PUT.
- **TRACE** : permet au client d'avoir en réponse à une requête, un retour des informations reçues par le serveur. Cela signifie que dans le corps de la réponse, le serveur inclut les entêtes qu'il a reçus dans la requête émise par le client. Cette méthode est utilisée pour tester la connexion.
- **CONNECT** : ne fonctionne que sur des serveurs mandataires et permet au client de se connecter via un tunnel HTTP (ou HTTPS pour des données chiffrées).
- **LINK/UNLINK** : permet d'associer (et de dissocier) des informations de l'entête à un document sur le serveur.
- **PATCH** : Cette méthode est similaire au PUT à l'exception que l'entité contient une liste des différences entre la version originale et la ressource identifiée dans l'URI.

  Les trois dernières méthodes présentées sont rarement implémentées.

7.4.4 Les réponses HTTP

A chaque requête du client, le serveur retourne une réponse qui est composée de (*listing 7.2*) :

- la première ligne : contient la version utilisée du protocole, le code de retour, et l'information qui correspond à la définition en langage humain du code de retour.
- les lignes suivantes sont composées de différents champs et potentiellement de données.

Listing 7.2– Exemple de réponse HTTP

```
1 HTTP/1.1 200 OK
2 Server: HTTPd/1.0
3 Date: Sat, 27 Nov 2004 10:19:07 GMT
4 Content-Type: text/html
5 Content-Length: 10476
```

Les codes de retour

Le code de retour d'une réponse HTTP permet de donner des informations sur le bon déroulement de la communication. La norme du protocole HTTP/1.1 définit 5 classes de code de retour. Le tableau 7.2 récapitule l'ensemble de ces classes ainsi que les codes de retour les plus rencontrés.

Classe	Code	Information
1xx (Information)	100	Continue
	101	Switching Protocols
2xx (Succès)	200	OK
	201	Created
	206	Partial Content
3xx (Redirection)	301	Moved Permanently
	302	Found
	304	Not Modified
	307	Temporary Redirect
4xx (Erreur du client)	400	Bad Request
	401	Unauthorized
	403	Forbidden
	404	Not Found
	405	Method Not Allowed
	406	Not Acceptable
5xx (Erreur du serveur)	500	Internal Server Error
	501	Not Implemented
	503	Service Unavailable
	505	Version Not Supported

Table 7.2 – Les codes de retour d'une réponse HTTP

7.4.5 Les champs d'entête

Certains entêtes peuvent se trouver aussi bien dans la requête que dans la réponse :

- `Content-Length` : la taille en octet des données utiles.
- `Content-Type` : type du contenu transmis ainsi que le jeu de caractères utilisé.
- `Connection` : options souhaitées pour la connexion courante.

Les entêtes suivants n'existent que dans les requêtes HTTP. Seul l'entête `Host` est cependant obligatoire dans la version 1.1 de HTTP :

- `Host` : spécifie le nom et optionnellement le port de l'hôte qui héberge la ressource demandée ;
- `User-Agent` : indique le programme client utilisé pour émettre la requête ;
- `Accept` : spécifie les types de supports qui sont acceptés ;
- `Accept-Language` : définit les langues acceptées par le client ainsi que leurs facteurs de priorité ;
- `Accept-Encoding` : similaire au champ `Accept`, mais apporte une restriction sur le codage du contenu ;
- `Accept-Charset` : indique les jeux de caractères acceptables ;
- `Keep-Alive` : temps maximal d'attente de la réponse et nombre maximum de requêtes par connexion ;
- `Cookie` : données de cookie mémorisées par le client ;
- `Referer` : URL de la page à partir de laquelle le document est demandé ;
- `Cache-Control` : spécifie les directives qui doivent être respectées par tous les mécanismes de mise en cache.

Et parmi les entêtes d'une réponse, il existe :

- `Content-encoding` : définit la méthode d'encodage des données envoyées ;

- `Content-language` : langue dans laquelle le document retourné est écrit ;
- `Date` : date et heure courante du serveur ;
- `Expires` : date et heure après laquelle la réponse est considérée comme expirée ;
- `Last-modified` : date de dernière modification du document envoyé ;
- `Location` : adresse du document lors d'une redirection ;
- `Etag` : version du document ;
- `Server` : nom et version du logiciel serveur.

7.4.6 Accès aux ressources protégées

Lorsqu'un client demande une ressource protégée, il envoie une requête GET au serveur en indiquant l'URL de celle-ci. Le serveur répond par un code 401 (*Authorization Required*) si toutes les conditions d'authentification ne sont pas remplies. La réponse contient l'entête `WWW-Authenticate` qui indique la méthode d'authentification utilisée. Le protocole HTTP supporte deux types de méthodes d'authentification :

- la méthode *basic* : cette méthode n'est pas sécurisée car les données transmises ne sont pas chiffrées. Elles sont simplement codées avec l'algorithme `Base64`.
- la méthode *digest* : utilise le haché du mot de passe transmis au serveur. Même si cette méthode est plus sûre que la méthode *Basic*, elle reste tout de même sensible aux attaques (vol de fichier de mot de passes).

Quand le client reçoit une réponse 401, il renvoie une nouvelle requête GET contenant le champ `Authorization` dont la valeur indique la méthode d'authentification utilisée (*Basic* ou *Digest*) ainsi que les valeurs des champs du formulaire d'authentification.

Une authentification réussie est suivie d'une réponse avec un code de retour 200, incluant la ressource demandée, sinon le code d'erreur 401 est renvoyé.

7.4.7 Antémémoire

Les mécanismes de base d'antémémoire ou mémoire cache dans HTTP consistent à sauvegarder temporairement dans un cache les ressources demandées dans des requêtes. Cette stratégie permet de réduire le nombre d'accès au serveur et les envoie répétés d'une même ressource à chaque requête. Il existe différents niveaux de mémoire cache :

- cache local : un client peut être amené à demander souvent les mêmes pages Web. Les pages Web chargées sont stockées dans le cache durant un temps fixé par le client. A un nouvel accès à une page, si cette page se trouve déjà dans le cache et qu'elle est encore « valide », le client s'en sert directement. Cette technique peut limiter les transferts sur le réseau.
- Server Cache : est un cache intermédiaire entre le client et le vrai serveur. Il communique avec le vrai serveur et transmet les réponses au client tout en gardant une copie de celles-ci (cache). Si un client demande au serveur-cache une page qui se trouve déjà dans son cache et qu'elle est encore "valide", il la transmet au client. Cette approche permet de diminuer la charge sur le serveur et le temps d'obtention d'une page.
- Server Proxy : Le serveur proxy est un intermédiaire entre le client et le serveur. Il sert de pare-feu. Il interroge le vrai serveur à la place du client et sert de cache.

Dans la version HTTP 1.1, l'entête `Cache-Control` a été ajoutée offrant la possibilité de définir des directives explicites aux caches HTTP. L'entête `Cache-Control` permet à un client ou serveur de transmettre diverses directives dans des requêtes ou des réponses. Ces directives outrepassent normalement les algorithmes de mise en antémémoire par défaut. En règle générale, si il y a un conflit apparent entre les différentes valeurs d'entête, c'est l'interprétation la plus restrictive qui s'applique. Les directives suivantes peuvent être appliquées :

- `no-store` : empêche le stockage non volatile des données ;
- `max-age` : permet au serveur de fixer la durée maximale de rétention ;
- `no-transform` : indique au cache qu'il ne doit pas transformer le corps du message qu'il reçoit ;
- `max-stale` : permet à un client d'autoriser le(s) cache(s) à renvoyer une réponse expirée, tout en fixant une limite à cette péremption ;
- `min-fresh` : permet à un client d'exiger une réponse qui sera valable pendant toute la durée indiquée.

7.5 Conclusion

Dans ce chapitre nous avons présenté le protocole HTTP et les différentes techniques de validation de protocoles, particulièrement le *fuzzing*. L'étude approfondie de la spécification RFC 2616 du protocole HTTP, nous a permis de comprendre en détail son fonctionnement et le rôle de chaque champ composant les requêtes et réponses. La complexité et le nombre d'information important que peuvent véhiculer les requêtes/réponses HTTP peuvent conduire à des vulnérabilités dues au non-respect de la spécification ou des failles de sécurité dans son implémentation. Nous présentons dans le chapitre suivant notre outil de *fuzzing*, permettant de vérifier la conformité et la robustesse du protocole HTTP embarqué dans les cartes à puce.

Chapitre 8

Fuzzing HTTP

Sommaire

8.1 Présentation générale **117**
8.2 L'application *PyHAT* **118**
8.2.1 Présentation 118
8.2.2 Fonctionnalités de *PyHAT* 119
8.2.3 Stratégie d'analyse 120
8.3 L'outil *Smart-Fuzz* **123**
8.3.1 Présentation 123
8.3.2 Étude des requêtes HTTP 123
8.3.3 Conception d'une BNF (*Backus Normal Form*) HTTP interactive 128
8.3.4 Modélisation des données de test 128
8.3.5 Configuration de Peach pour les cartes à puce SCWS 129
8.3.6 Le parallélisme 130
8.3.7 Fichiers de journalisation 130
8.3.8 Analyse des événements 131
8.4 Conclusion **132**

Dans ce chapitre nous présentons notre deuxième contribution dans le cadre de cette thèse, qui porte sur la sécurité des cartes à puce à serveur Web embarqué, au niveau du protocole HTTP implémenté. Les failles d'implémentation dans un protocole peuvent être dues au non respect de la spécification et conduire à des vulnérabilités. L'implémentation du protocole HTTP doit respecter la spécification définie dans RFC 2616 [FGM+99] et être robuste et sécurisée en garantissant que des commandes sensibles (ajout, suppression, modification) ne puissent pas s'exécuter dans un autre contexte que celui autorisé. Pour vérifier la conformité et la robustesse de l'implémentation du protocole HTTP embarqué dans les cartes à puce, nous avons choisi d'utiliser la technique du *fuzzing*.

8.1 Présentation générale

L'outil que nous avons mis en œuvre permet de vérifier l'implémentation du protocole HTTP dans un serveur Web quelconque. Nous nous intéressons particulièrement aux serveurs web embar-

qués dans des cartes à puce. Nous n'avons aucune connaissance sur l'implémentation du serveur ciblé ; notre outil fonctionne en boite noire, en se basant sur la spécification HTTP.

Notre outil est composé de deux applications complémentaires *PyHAT*(*Python HTTP Assessment Test*) et *Smart-Fuzz* (*figure : 8.1*). Le *fuzzing* est réalisé par l'application *Smart-Fuzz* qui se base sur le logiciel Peach. Les données de test sont générées de manière semi-aléatoire en utilisant un ensemble de descripteurs de données (les modèles de données, les modèles d'états ainsi que les mutations) représentant les requêtes HTTP et les différents champs qui les composent. Afin de rendre le *fuzzing* plus intelligent et réduire le temps d'exécution excessif dû à un nombre important de données de test dont certaines seraient inutiles, nous avons développé l'application *PyHAT*. *PyHAT* est exécutée en premier. Elle permet de détecter les différentes fonctionnalités supportées par l'implémentation du protocole HTTP dans la carte à puce testée. Les résultats de cette application sont ensuite exploités par l'outil *Smart-Fuzz* qui limite la génération des tests à uniquement des fonctionnalités implémentées dans l'application ciblée.

Nous avons également configuré un *fuzzing* parallèle pour répartir les tests sur un ensemble de cartes à la fois et réduire le nombre de données traitées par une carte. Les résultats du *fuzzing* sont répartis dans des fichiers d'événements structurés qui sont ensuite automatiquement analysés pour détecter des vulnérabilités.

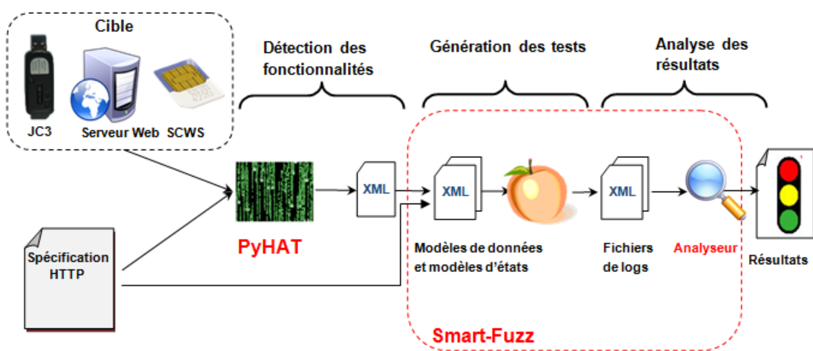

Fig. 8.1 – Vue générale de l'outil de *fuzzing* HTTP

8.2 L'application *PyHAT*

8.2.1 Présentation

PyHAT est un outil d'analyse de serveurs Web. Il permet de mettre en évidence les principales caractéristiques des serveurs HTTP installés sur différents supports. Il a été conçu dans un but d'analyser des serveurs embarqués dans des cartes à puce, mais il est cependant en mesure d'évaluer toute application implémentant le protocole HTTP du côté serveur (*figure : 8.2*). Le but principal de *PyHAT* est de recueillir autant d'informations que possible sur les fonctionnalités HTTP implémentées sur le serveur ciblé afin de réduire le nombre de tests à réaliser lors de l'étape du *fuzzing*.

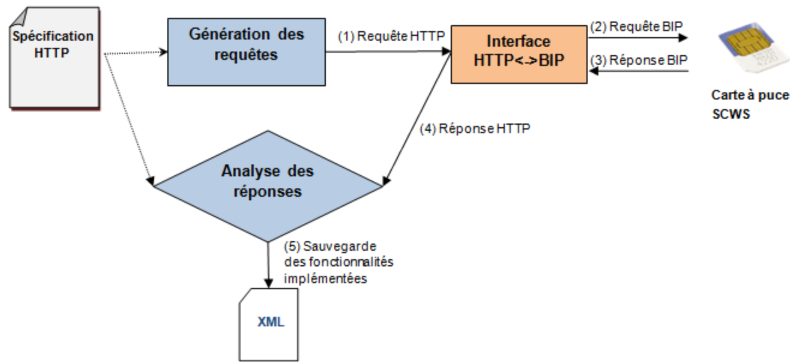

Fig. 8.2 – Fonctionnement de l'application *PyHAT* sur une carte SCWS

8.2.2 Fonctionnalités de *PyHAT*

Dans ce qui suit, nous présentons les différentes fonctionnalités HTTP que l'application *PyHAT* permet de détecter.

1. **Détection des méthodes HTTP implémentées**

   Comme vu dans le chapitre précédent, le protocole HTTP est principalement caractérisé par ses méthodes. Celles-ci permettent aux clients d'interagir avec les ressources du serveur et de spécifier l'action à effectuer. La norme du protocole définit 8 méthodes de base auxquelles peuvent se rajouter des extensions. La première fonctionnalité de *PyHAT* est de tenter de déterminer celles qui sont implémentées dans un serveur donné.

2. **Évaluation de la sensibilité à la casse des requêtes**
   La spécification du protocole HTTP indique que les méthodes et entêtes des requêtes envoyés au serveur doivent être sensibles à la casse. Les méthodes doivent être entièrement en majuscule (exemple : `POST`), tandis que dans les entêtes, uniquement la premiere lettre des mots le composant est en majuscule (exemple : `Transfer-Encoding`). Notre outil d'évaluation comporte une fonctionnalité qui permet de savoir si l'implémentation du serveur testé respecte cette propriété.

3. **Détection des versions supportées**

   Depuis sa création, le protocole HTTP a connu 3 versions majeures. *PyHAT* compte parmi ses fonctionnalités la détection des versions supportées dans le serveur ciblé.

4. **Détection des méthodes d'encodage de contenu supportées**

   Le protocole HTTP autorise l'encodage du contenu d'une page demandée en appliquant une méthode spécifiée par le client et supportée par le serveur. Parmi les méthodes d'encodage définies dans la RFC 2616, IETF, on cite : `gzip, compress, deflate`. Il est possible d'indiquer dans une requête une préférence de type d'encodage pour une ressource demandée. Le mot clé `identity` signifie que les données attendues ne doivent pas être encodées. *PyHAT* dispose d'une fonction lui permettant de détecter quels mécanismes d'encodage de données sont gérés par le serveur.

5. **Détections des entêtes HTTP implémentés dans le serveur**

   Nous avons présenté dans le chapitre précédent la structure d'un paquet HTTP. Outre la première ligne concernant la requête effective, plusieurs autres champs (entêtes) en rapport avec la ressource ou encore les capacités du client, peuvent être intégrés dans ce paquet. *PyHAT* permet de détecter les différents champs implémentés.

6. **Récupération d'une liste des pages du serveur**

   PyHAT compte parmi ses fonctionnalités celle de récupérer une liste des URI présentes sur le serveur à partir d'une analyse de la page « index » se trouvant à la racine de ce dernier.

7. **Synthèse des résultats dans un fichier au format XML**

   L'application *PyHAT* génère une synthèse de l'ensemble des résultats obtenus durant le processus d'évaluation dans un fichier au format XML, ce fichier est utilisé par l'application *Smart-Fuzz* dans la génération des données de test.

8.2.3 Stratégie d'analyse

La stratégie appliquée dans *PyHAT* pour fournir les fonctionnalités listées précédemment consiste à transmettre au serveur ciblé des requêtes bien spécifiées et de déduire en fonction des réponses HTTP, si une fonctionnalité est implémentée ou pas (*figure 8.2*).

Comme présenté dans le chapitre précédent, le protocole HTTP définit un ensemble de codes de retour, chacun ayant une signification particuliere. Ces codes de retour ont été très utiles pour déterminer les caractéristiques du serveur testé. *PyHAT* est conçu pour être appliqué sur tout serveur HTTP, quel que soit son environnement d'exécution (standard, carte à puce, etc.). Cependant dans le cas des cartes à puce SCWS, nous définissons une interface HTTP<->BIP qui se place entre le générateur des données de test (requêtes HTTP) et le serveur SCWS. Son rôle consiste à encapsuler les paquets d'une requête HTTP dans un format BIP supporté par le SCWS. Inversement, les réponses BIP reçues par le serveur sont décapsulées afin de récupérer la réponse HTTP qui sera analysée (*figure 8.2*). Dans ce qui suit, nous présentons la stratégie adoptée pour l'implémentation des différentes fonctionnalités de *PyHAT*.

Cas des méthodes implémentées : la RFC 2616 indique qu' « un serveur DEVRAIT retourner le code d'état 501 (Non mis en œuvre) si la méthode n'est pas reconnue ou pas mise en œuvre dans le serveur d'origine ». Nous avons exploité cette propriété pour déterminer les méthodes implémentées dans un serveur. En pratique, pour chaque méthode, nous avons défini une requête respectant le format de la norme HTTP présenté dans le chapitre précédent. Seule la première ligne de la réponse nous intéresse. En effet, à partir de celle-ci nous pouvons isoler le code de retour et déterminer si la méthode concernée est implémentée ou pas.

Tester la sensibilité à la casse : le test de la sensibilité à la casse porte à la fois sur le nom de la méthode et les champs de la requête. Pour réaliser ces tests, nous définissons dans un premier temps des requêtes dont la méthode utilisée est écrite en minuscule. Dans un second temps, la méthode est formatée conformément à la norme (en majuscule), mais les autres champs sont écrits de façon aléatoire. Dans chacun des cas, l'analyse de la réponse se fait à

base du code de retour. En effet, si le serveur répond avec un code 200 (OK) alors nous en déduisons que ce dernier ne fait aucune distinction entre les majuscules et minuscules.

Cas des versions supportées : évaluer quelles sont les versions du protocole supportées par un serveur est simple. Lors de l'envoi d'une requête à un serveur, ce dernier doit si c'est possible répondre avec la même version que celle utilisée par le client. Notre analyse se base donc sur ce fait et suit l'algorithme suivant :

- envoi d'une requête dans une version donnée ;
- réception de la réponse et analyse de la version utilisée ;
- si la version de la requête est la même que celle indiquée dans la réponse, alors celle-ci est supportée par le serveur ;
- sinon, la version utilisée dans la requête n'est pas supportée.

Cet algorithme est répété pour chacune des versions.

Remarque : La norme définit un code de retour correspondant à une version non prise en charge (505). Cependant, nous avons remarqué que certains serveurs n'en tiennent pas compte. C'est pourquoi nous avons choisi de suivre l'algorithme précédent.

Cas des encodages supportés : Le serveur doit répondre à une requête en utilisant lorsque c'est possible un mécanisme d'encodage supporté par le client sinon aucun encodage n'est réalisé. D'un point de vue pratique, lorsqu'un client envoie une requête, il spécifie quels sont les encodages qu'il supporte via le champ `Accept-Encoding`. Le serveur qui reçoit cette requête commence par analyser ce champ. Si la valeur de ce dernier fait partie des mécanismes d'encodage qu'il peut gérer alors la réponse inclura le champ `Content-Encoding` dont la valeur sera la même que l'encodage utilisé dans la requête.

La norme définit 4 valeurs possibles pour ces champs : gzip, compress, defate, identity.

Afin de déterminer les encodages gérés par un serveur, *PyHAT* envoie une requête par type d'encodage et analyse le champ `Content-Encoding` de la réponse. Il existe deux cas possibles :

- la réponse ne contient pas le champ `Content-Encoding`. Dans ce cas, deux interprétations sont possibles :
  1. l'encodage n'est pas géré ;
  2. la valeur de l'encodage dans la requête est `identity`.
- la valeur du champ `Content-Encoding` est la même que dans la requête. Nous en déduisons alors que l'encodage est géré.

Cas des champs supportés : la détection des champs implémentés dans un serveur est de loin la tâche la plus complexe à réaliser. En effet, la spécification HTTP ne fournit pas de règles explicites permettant de déterminer pour l'intégralité des champs si ceux-ci sont supportés.

La stratégie que nous avons adoptée est celle d'envoyer des requêtes respectant le format définit dans la norme avec un champ spécifié par la requête. En outre, en fonction de la valeur du champ, nous savons d'après la RFC 2616 à quelle réponse du serveur nous devons nous attendre. Pour les champs dont aucune indication n'est présentée dans la norme, nous considérons par défaut qu'ils sont implémentés. Le tableau 8.1 présente quelques exemples de champs et la stratégie utilisée pour vérifier leur implémentation. Pour chacun de ces champs nous avons injecté une valeur qui selon la spécification devrait retourner un code d'erreur particulier.

Id	Champ	Valeur injectée	Réponse attendue
1	Accept	text/plain ;q=1,text/html ;q=0	code 406
2	Accept-Charset	ISO-8859-1,utf-8 ;q=0	code 406
3	Expect	10-continue	code 4xx
4	If-Modified-Since	Sun, 06 Nov 2014 08 :49 :37 GMT	code 200
5	If-Range	UUID généré aléatoirement	code 200
6	If-Unmodified-Since	Sun, 06 Nov 1970 08 :49 :37 GMT	code 412
7	Range	1-30	code 206
8	Content-Length	100000000000000000000000000	code 413

Table 8.1 – Stratégie de détection des champs HTTP implémentés

Les lignes du tableau 8.1 sont expliquées respectivement ci-dessous.

1. Interdire la réception de contenu HTML au profit de contenu de type « text » devrait provoquer au niveau du serveur l'envoi d'une réponse indiquant son incapacité à fournir ce type de données (code 406).
2. Si le serveur ne peut pas fournir au client une réponse qui applique un encodage spécifié par le client, il devrait renvoyer une réponse de type 406.
3. Une valeur du champ `Expect` qui ne correspond pas à ce qui est défini dans la norme devrait engendrer une erreur de classe 4xx.
4. Si la ressource demandée n'a pas été modifiée depuis la date indiquée, le serveur doit retourner un code 304. Dans notre cas, la date est invalide (supérieure à l'année en cours) ; le serveur devrait alors retourner un code 200.
5. La valeur utilisée pour le champ `if-Range` est générée aléatoirement, il est peu probable qu'elle corresponde à l'étiquette d'une entité présente sur le serveur. Le code attendu est donc 200.
6. Si une ressource n'a pas été modifiée depuis l'heure indiquée, le serveur doit retourner un code 412. La date que nous avons fixée étant très ancienne par rapport à la date courante, il est très peu probable que la ressource demandée n'ait pas été modifiée depuis celle-ci.
7. La demande de données partielles doit déboucher sur une réponse de type 206. Si ce n'est pas le cas, la partie demandée n'est pas accessible ou le champ n'est pas supporté.
8. Si un serveur reçoit une demande avec un contenu de taille supérieure à ce qu'il peut gérer, il doit retourner un code 413.

Cas de l'URI Tracker : la récupération de l'ensemble des URI d'un site donné suit le principe de fonctionnement des aspirateurs de sites classiques.

En partant de la page « index » du site, nous récupérons toutes les chaînes de caractères correspondant à une certaine expression régulière (dans notre cas, celle des attributs `href`). Nous recommençons l'opération sur chacune des URI trouvées en prenant soin d'effectuer quelques vérifications :

– nous vérifions que l'URI détectée n'est pas déjà présente dans la liste des résultats afin d'éviter de boucler indéfiniment.

– dans le cas où une URL est récupérée (domaine/ressource) nous vérifions que le domaine a la même adresse IP que le domaine analysé initialement.

8.3 L'outil *Smart-Fuzz*

8.3.1 Présentation

Smart-Fuzz constitue le cœur de notre outil de *fuzzing* HTTP, il est composé principalement de 4 parties : génération des données de test, transmission des données à la cible, génération des fichiers d'événements, analyse des résultats (*figure 8.3*). Il est conçu pour tester tout type de serveur Web, standard ou embarqué. Nous nous intéressons particulièrement au cas des serveurs SCWS.

Son objectif est de vérifier la conformité et la robustesse du protocole HTTP implémenté dans un serveur Web en analysant son comportement quand il reçoit des entrées invalides (crash, comportement inattendu de la carte, comportement non conforme à la spécification).

Pour générer les données de test, *Smart-Fuzz* se base sur le logiciel Peach. Il utilise des descripteurs de requêtes HTTP prédéfinis et limite les tests sur uniquement des fonctionnalités supportées par la cible et qui sont spécifiés dans le fichier résultat de *PyHAT*.

Smart-Fuzz est également composé d'une interface HTTP<->BIP optionnelle, utilisée dans le cas de test sur des serveurs SCWS, pour garantir la compatibilité du format des données de test avec la cible.

La détection des vulnérabilités se base sur l'analyse des codes de retour, à savoir : le code de retour BIP, le code de retour APDU *status word* contenu dans la réponse BIP et le code de la réponse HTTP en fonction des données envoyées en entrée. Nous avons également appliqué un *fuzzing* parallèle pour optimiser le temps de *fuzzing*.

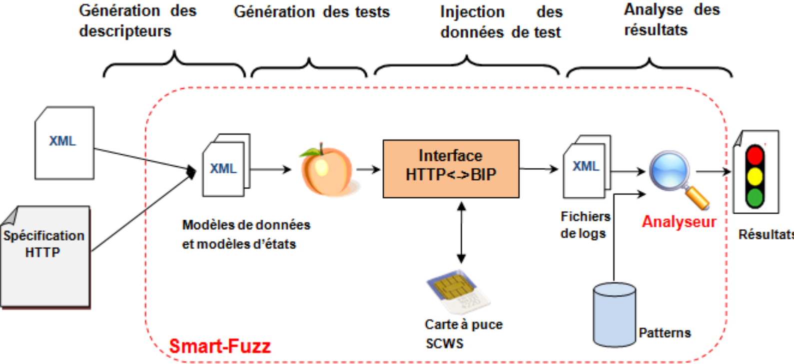

Fig. 8.3 – Fonctionnement de l'application *Smart-Fuzz* (cas des serveurs SCWS)

8.3.2 Étude des requêtes HTTP

La construction d'un modèle représentatif des requêtes HTTP nécessite une étude exhaustive de ces dernières. Nous nous sommes basés sur la RFC 2616 pour étudier de manière détaillée les différents champs qui composent une requête, en tenant compte de la version du protocole HTTP.

La stratégie de notre outil de *fuzzing* consiste à muter un champ par requête injectée et d'analyser les réponses du serveur et de la carte comparées aux réponses attendues.

Telle que présentée dans le chapitre précédent, une requête HTTP est composée de 3 niveaux, une ligne de commande, une ou plusieurs lignes correspondant aux entêtes et optionnellement le corps de la requête :

```
<Requête> = <Method> <URI> <Version> CRLF
*(<Header>CRLF)
 CRLF
[<Corps>]
```

La ligne de commande

La ligne de commande est composée des champs : <Method>, <URI>, <Version>, et CRLF

- <Method> : l'ensemble des valeurs de ce champ est bien défini. Il peut prendre comme valeur un des types supportés par la plateforme testée (détectés par *PyHAT*). Exemple : GET, POST.
- <URI> : définit le chemin absolu ou relatif (à partir du serveur) qui localise la ressource demandée dans la requête.
- <Version> : définit la version du protocole HTTP qui sera utilisée durant la communication (utilisé uniquement dans les versions supérieures à HTTP/0.9). Sa syntaxe est de la forme : HTTP/<major>.<minor>, <major> et <minor> étant des entiers.
- <Header> : permet au client d'informer le serveur sur ses capacités (langage accepté, encodage accepté, etc. . .). Elle doit contenir le nom de l'entête suivi de sa valeur, les deux parties étant séparées par deux points comme suit :
  <champ> : *(<valeur>)
- <Corps> : est utilisé pour acheminer les données définies par l'utilisateur. Il est représenté sous forme d'une chaine de caractères aléatoire.

Les entêtes HTTP

Une requête HTTP peut être composée de différents champs (entêtes) dont le format est différent les uns des autres. Étant donnée la quantité d'informations importante contenue dans la spécification du protocole HTTP, nous nous limitons dans ce qui suit à la description de quelques exemples de champs que nous avons modélisés.

1. **Format de date et d'heure**
   Par souci de compatibilité avec les versions antérieures du protocole, nous tenons compte des différents formats de date et d'heure définis dans chaque norme. Les trois différents formats existants sont :
   - Sun, 06 Nov 1994 08 :49 :37 GMT
   - Sunday, 06-Nov-94 08 :49 :37 GMT
   - Sun Nov 6 08 :49 :37 1994

   Un serveur HTTP/1.1 doit prendre en compte ces trois différents formats.

2. **Caractéristiques du contenu**

   Le protocole HTTP permet de négocier les caractéristiques des ressources attendues (encodage, type, langue). Il existe différents formats de codage de contenu (*gzip, compress, deflate*, etc). Tous ces formats ont été définis dans le registre de l'IANA (*Internet Assigned Number Authority*). Ils peuvent être demandés par le client dans le champ `Accept-Charset` ou dans le champ `Content-Encoding`. Par défaut, le codage `Identity` (aucun codage) est appliqué. L'entête `Accept` définit le type attendu : elle prend comme valeur un des types MIME ou le caractère * pour spécifier que le client accepte tous les types/sous-types. L'entête `Accept-Language` spécifie les différentes langues acceptées.

3. **Codage de transfert**

   C'est le type de codage appliqué à une entité pour assurer un transport sécurisé sur le réseau. Contrairement au codage de contenu, c'est une propriété du message en lui-même. Il est défini par l'entête `Transfer-Encoding`. La valeur `chuncked` définie par la RFC 2616 pour ce champ permet de transmettre une ressource en une série de fragments. Chaque fragment commence par un nombre représentant la taille en octet du fragment envoyé. Si le nombre envoyé est 0, cela signifie l'atteinte de la fin du message.

   Il existe des entêtes spécifiques au cas du codage de transfert `chuncked`. Comme le message est généré automatiquement, il est possible que certains entêtes ne soient connus qu'après la génération du message. L'entête `Trailer` permet de définir les différents entêtes qui seront envoyés à la fin du message. L'entête `TE` est utilisé pour définir quels codages de contenu (`content-encoding`) seront acceptés par le client et si l'entête `Trailer` est supporté par le client, le mot clé `trailers` est ajouté à la liste des encodages.

4. **Valeurs de qualité**

   Certains champs, notamment le champ `Accept`, utilisent des valeurs de qualité permettant de préciser un ordre de priorité des préférences du client, concernant le type et l'encodage des données attendues en réponse. Ces valeurs sont comprises entre 0.000 et 1.000 avec trois digits au maximum après le point. La valeur 0.000 signifie que le type n'est pas accepté alors que la valeur 1.000 signifie que c'est la préférence maximale. La valeur la plus forte compte toujours en premier. Si aucune valeur de qualité n'est précisée, alors il s'agit de la préférence maximale.

   Exemple : `Accept: text/plain; q=0.5, text/html text/x-dvi; q=0.8, text/x-c`

   Cet exemple est interprété comme suit : `"text/html` et `text/x-c"` sont les types de données préférés par le client, mais s'ils ne sont pas supportés par le serveur alors il faut utiliser des entités `text/x-dvi` et si ce dernier n'est également pas supporté alors il faut utiliser le type `text/plain`.

5. **Étiquettes d'entités**

   Elles sont utilisées dans les entêtes `E-tag, If-Match, If-None-Match et If-Range, Range` pour comparer les versions de deux ou plusieurs entités reçues après plusieurs demandes d'une même ressource. Elles sont utilisées dans la gestion des caches pour vérifier si la page a changé depuis la dernière requête. L'étiquette d'une entité est une chaîne de caractères présentée entre guillemets avec éventuellement un préfixe indicateur de faiblesse. L'indicateur fort (`Strong Validators`) est partagé entre deux ressources si elles sont identiques octet par octet et l'indicateur faible (`Weak Validators`) noté par `"W/"`, correspond à vérifier simplement que les

ressources sont sémantiquement équivalentes ; il est utilisé pour alléger les traitements sur certains serveurs. Exemple :

`ETag: W/"123456789"` étiquette à indicateur faible.

`ETag: "123456789"` étiquette à indicateur fort.

6. **Transactions partielles**

   Ce processus a été introduit dans HTML/1.1. Il consiste à découper la ressource en parties (en octets). Le client spécifie la partie de la ressource qu'il souhaite récupérer dans l'entête `Range`. La valeur de cet entête peut être un intervalle d'octets borné (de l'octet 200 à l'octet 400) ou infinie (à partir du 100 ème octet), ou consister en un ensemble de parties demandées dans une même requête.

   Exemple : `Range : bytes=100-1000` intervalle d'octets borné ("de l'octet 100 à l'octet 1000")

   La réponse sera donc segmentée, contenant les parties demandées. Si la requête n'est pas satisfaite le code de réponse `416` sera retourné. La réponse contient l'entête `Accept-Range` qui indique que l'unité de mesure "octet" est acceptée par le serveur et l'entête `Content-Range` indique l'intervalle d'octets transmis et la taille totale de la ressource. Les serveurs qui implémentent les réponses partielles renvoient la partie demandée avec le code `206` (*partial content*) sinon le champ `Range` est ignoré et la requête est traitée normalement.

7. **Le type Multipart**

   Il procure un mécanisme commun pour représenter des objets composés d'un ensemble de parties distinctes de types MIME. Tirés du format MIME, les types `multipart` permettent d'encapsuler un ou plusieurs corps dans un même corps. Comme dans le format MIME, une délimitation `boundary` est utilisée pour séparer les différents corps. Chaque corps peut avoir différents entêtes qui renseignent sur les caractéristiques d'une partie.

   Exemple :

```
Content-type: multipart/byteranges; boundary="content_example"
type=Text/HTML; start=example1

--content_example
Part 1:
Content-Type: Text/HTML; charset=US-ASCII
Content-ID: <example1>
Content-Location: http://www.example.com/images/the.one

--content_example
Part 2:
Content-Type: Text/HTML; charset=US-ASCII
Content-ID: <example2>
Content-Location: the.one ;
Content-Base: http://www.example.com/images/
--content_example--
```

   L'entête `Content-Type` définit le type et les sous-types des données (exemple `Text/HTML`) et exceptionnellement un paramètre `charset` définissant le jeu de caractères utilisé et séparé du type de données par un point-virgule. La valeur de l'entête `Content-ID` est une chaine de caractères identifiant une partie de la ressource.

8. **Les requêtes conditionnelles**

   Une requête peut être rattachée à des conditions indiquées dans les champs `If-Match, If-None-Match, If-Modified-Since, If-Unmodified-Since, If-Range`. Un de leurs cas d'utilisation est en rapport avec la gestion du cache. Dans le cas où la page demandée a déjà été mise dans le cache, une validation est effectuée pour savoir si la ressource du serveur a changé entre temps. Si la ressource n'a pas changé, le serveur ne va pas renvoyer la ressource mais juste les entêtes de la réponse, ce qui améliore la performance du réseau. Le tableau 8.2 présente le type de valeurs attendues pour chacune des différents entêtes :

Entête	**Format de la valeur**
`If-Match`	"étiquette d'entité"
`If-None-Match`	"étiquette d'entité"
`If-Modified-Since`	HTTP-date
`If-Unmodified-Since`	HTTP-date
`Last-Modified`	HTTP-date
`If-Range`	"étiquette d'entité" | HTTP-date

Table 8.2 – Les requêtes conditionnelles

9. **La persistance de la connexion**

   Dans les premières versions du protocole HTTP, une connexion TCP était ouverte et fermée pour chaque bloc requête/réponse, ce qui posait des problèmes de congestion du réseau. Dans le Web, une ressource est presque toujours rattachée à d'autres. Par exemple une page Web est rattachée à ses feuilles de style, ses scripts, ses images etc. Ce qui équivaut à parfois bien plus d'une dizaine de requêtes GET. Si on multiplie cela par le nombre de pages demandées par un client et le nombre de clients simultanés pour un serveur, les connexions TCP sont bien trop nombreuses et surchargent le serveur inutilement.

   Les connexions TCP sont devenues par défaut persistantes dans la version HTTP/1.1, en ajoutant le champ `Connection` et sa valeur `keep-alive` qui force la connexion TCP à rester ouverte. Le navigateur peut alors relancer une nouvelle requête sur une même connexion, sans avoir à rétablir un nouvel échange TCP. La valeur `close` est réservée pour spécifier que la connexion doit être fermée après traitement de la requête en cours.

10. **Mise en place d'un champ `HOST`**

    Avant la version HTTP/1.1, il n'existait pas de solution standard pour faire face aux problèmes de confusion que rencontraient les serveurs hébergeant plusieurs sites Web de nom de domaine différents et recevant une requête avec un chemin relatif. D'où la mise en place de ce champ OBLIGATOIRE (le seul dans la norme HTTP) indiquant le nom du domaine réel et virtuel du serveur. Pour des soucis de rétrocompatibilité avec les anciennes versions, des règles précises doivent être respectées pour des requêtes HTTP/1.1. En effet, le domaine peut également être présent dans le champ `<Request-URI>` si l'URI est absolue et porter confusion avec le champ `HOST`. Ces règles donnent la priorité au champ `<Request-URI>`,

    – si ce champ contient le nom du domaine, le champ `HOST` doit être ignoré,
    – si le nom du domaine pris en compte est invalide ou le champ `HOST` est absent, le code de statut `400 (Bad request)` est renvoyé,
    – Le champ `HOST` peut-être vide, il indique alors le même nom de domaine que la couche de transport (TCP).

8.3.3 Conception d'une BNF (*Backus Normal Form*) HTTP interactive

Étant donnée la masse importante d'informations contenues dans la RFC 2616, il nous est paru utile de créer un outil permettant de visualiser et de retrouver facilement et rapidement celles-ci. Nous avons implémenté une interface développée en HTML et JavaScript. Elle se présente sous forme de BNF interactive composée par une structure de liens où le plus haut niveau d'une requête ou d'une réponse HTTP est représenté et chaque sous partie de cette dernière se développe à l'aide de clics. L'utilisateur peut ensuite rajouter les entêtes qu'il souhaite.

La description de chaque entête HTTP a été représentée par 5 caractéristiques :

- **Do** : explique le rôle de l'entête dans la requête/réponse,
- **Compatibility** : versions HTTP compatibles avec celui-ci,
- **Constraints** : spécifie les fortes contraintes associées à celui-ci,
- **Warning** : spécifie certaines situations et traitements particuliers,
- **Status-code** : indique les codes de retour HTTP possibles pour cet entête, et pour quelle valeur chaque code est retourné.

L'exemple ci-dessous nous montre la description du champ « Date » :

```
"Date" ":" <HTTP-date> CRLF   [Hide Information]

Do : represents the date at wich the message has created
Constraint :
        - origin server MUST include this field in all response except in these cases:
                1. Status-Code = 100 | 101 => MUST turn in MAY
                2. Status-Code = 5XX
                3. If the server doesn't have a reliable clock, it MUST NOT include this field
                  and assignes an Expires or a Last-Modified field unless the values are bind
                  to an other reliable clock

        - client SHOULD only sent this field when the message include an entity-body (like PUT and POST)
           except with a not reliable clock

Compatibility : 1.0/1.1
```

Fig. 8.4 – BNF de l'entête Date

8.3.4 Modélisation des données de test

Les données de *fuzzing* doivent respecter une logique de programmation du système ciblé afin d'éviter que celui-ci les rejette dès qu'elles ne correspondent pas à ce qui est attendu. *Smart-Fuzz* est basé sur le logiciel Peach qui génère des données à partir d'un ensemble de fichiers descripteurs des formats de données valides et applique des mutations sur les différents champs composant un descripteur.

Compte tenu du nombre important d'informations contenues dans la RFC, nous utilisons la BNF que nous avons conçue pour la construction des fichiers descripteurs. Ces fichiers sont au format XML contenant un ensemble de structures de données. Ils sont générés par *Smart-Fuzz* et utilisés dans Peach pour la génération des différentes données de test. *Smart-Fuzz* se base sur le résultat de l'analyse de *PyHAT*, présent dans un fichier nommé *assessment.xml*, qui comporte toutes les fonctionnalités supportées (les méthodes, les champs, les encodages, etc.) dans la plateforme ciblée. Seules les propriétés définies dans le fichier *assessment.xml* sont modélisées et transmises à Peach. Il existe principalement trois types de structures de données :

Modèle de données : une requête HTTP est représentée par l'ensemble des champs qui la composent. Les champs utilisés sont différents selon le type de requête. Un modèle de données est une représentation en XML de la structure des champs composant la requête. Il consiste également à définir quels sont les éléments que nous voudrions ou ne voudrions pas muter et de fixer des valeurs par défaut dans certains cas. Peach offre la possibilité de répartir la description des données par blocs. Cette particularité offre l'avantage de réduire la quantité de modèle à générer. En effet, chaque entête est représenté par son modèle, et pour modéliser une requête il suffit de réunir les différents modèles correspondant aux entêtes contenus dans la requête.

Modèle d'états : il permet de constituer une séquence d'instructions dans Peach, correspondant aux flux de données pendant le processus de *fuzzing*. En effet, dans certains cas, la réponse du système peut dépendre d'une suite d'actions exécutées dans un ordre bien défini. Un modèle d'états doit contenir au moins un état.

Mutations : en plus de la représentation des données par leur structure et leur ordre d'application, nous avons défini les types de mutations à utiliser. En effet, certains champs n'acceptent que des types de données spécifiques tels que le type `String`. Le format de ces données peut également être fixé comme dans le cas des dates ou des URI. L'utilisation de cette approche permet de réduire considérablement le nombre de données de test utilisées par l'outil et d'éviter des tests inutiles avec des données non conformes.

8.3.5 Configuration de Peach pour les cartes à puce SCWS

À partir des fichiers XML (pit files) que nous avons définis, Peach construit les données HTTP à injecter dans la cible. Cependant dans le cas des cartes à puce SCWS, basées sur le protocole BIP, la carte s'attend à recevoir des paquets BIP encapsulant des paquets HTTP et Peach ne supporte pas le protocole BIP. Afin de palier à ce problème nous avons utilisé une application que nous appelons « interface HTTP<->BIP ». Cette interface joue le rôle d'un intermédiaire. Elle permet d'envelopper des paquets de données au format HTTP dans un format BIP accepté par la carte à puce. Les réponses BIP sont également traitées par l'interface HTTP<->BIP, pour ressortir la partie HTTP de la réponse, utile pour l'analyse des résultats.

Nous avons configuré Peach afin de désigner l'interface HTTP<->BIP comme cible du *fuzzing*. Nous avons spécifié un chemin d'accès vers cette interface qui reçoit les données de *fuzzing* pour les transférer par la suite à la carte à puce dans un format BIP (*figure 8.5*).

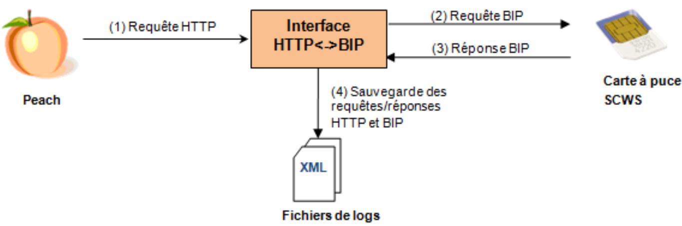

Fig. 8.5 – *Fuzzing* sur SCWS

8.3.6 Le parallélisme

L'inconvénient majeur du *fuzzing* est le temps important nécessaire avant la fin de l'analyse. Le *fuzzing* peut dans certains cas prendre plusieurs jours avant d'arriver au résultat final. Ce temps est principalement relatif au nombre de données de test appliquées au système.

L'utilisation des descripteurs de données (modèles de données, modèles d'états) et des mutations permet de réduire considérablement le nombre de données de test à générer, comparé à un *fuzzing* aléatoire. L'exploitation de l'application *PyHAT* permet également d'éviter des tests inutiles sur des fonctionnalités non supportées.

Afin d'améliorer le temps d'exécution, nous avons exploité l'option du parallélisme offerte par Peach. En effet, Peach permet de configurer le *fuzzing* sur plusieurs plateformes à la fois. Nous avons exploité cette propriété afin de répartir les données de test sur plusieurs cartes. Nous avons configuré Peach de telle sorte à générer autant d'ensemble de données que de cartes disponibles. Peach va ensuite transmettre ces données aux différentes cartes (si les cartes sont basées sur du TCP/IP) ou vers l'interface HTTP<->BIP (si les cartes sont basées sur le protocole BIP) qui les retransmet aux cartes SCWS. Ainsi le temps d'exécution serait réduit à T/N, T étant le temps nécessaire pour le *fuzzing* sur une seule carte et N le nombre de cartes utilisées dans le *fuzzing* parallèle.

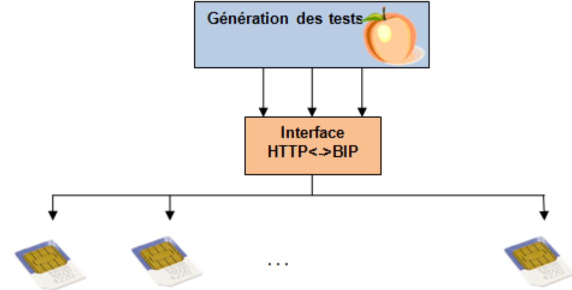

Fig. 8.6 – *Fuzzing* parallèle sur plusieurs cartes à puce

8.3.7 Fichiers de journalisation

Afin de vérifier la présence de failles ou de comportements anormaux dus au non-respect de la spécification du protocole HTTP, il est nécessaire d'analyser les différents messages envoyés et reçus par le serveur ciblé. Pour cela nous sauvegardons une trace des différentes transactions durant le *fuzzing* dans des fichiers d'événements.

Il existe deux possibilités de génération des fichiers d'événements :

- utiliser le mécanisme par défaut de Peach qui consiste à tout sauvegarder dans un fichier enregistré à la racine de Peach,
- ou concevoir un système de journalisation personnalisé.

Afin de faciliter l'analyse des résultats, nous avons choisi de définir notre propre organisation des fichiers d'événements. Dans le cas de cartes SCWS, le mécanisme de journalisation (*logging*) est géré par l'interface HTTP<->BIP qui transmet les données de *fuzzing* à la carte et reçoit les réponses du serveur. Les résultats sont répartis et organisés par types de méthode HTTP et par nom d'entête

muté dans la requête injectée. Nous définissons un ensemble de répertoires correspondant aux différentes méthodes supportées par l'application ciblée. Chacun de ces répertoires est composé de fichiers correspondant aux différents entêtes supportés. À chaque test, des informations concernant la requête et la réponse échangées entre *Peach* et la cible sont triées et sauvegardées dans le répertoire convenu. Les informations recherchées sont :

– le couple requête/réponse HTTP,
– la commande BIP,
– le code de retour HTTP,
– le code de retour de la carte « *status word* ».

Ces informations sont séparées par blocs requête/réponse.

8.3.8 Analyse des événements

L'analyse des résultats consiste à parcourir l'ensemble des fichiers d'événements à la recherche de comportements suspects ou non conformes à la norme HTTP. Pour la plupart des outils de *fuzzing* existants, cette tâche est effectuée manuellement par l'utilisateur. Ce dernier doit parcourir tous les résultats pour détecter une faille ou une anomalie. Afin de réduire l'intervention de l'utilisateur, nous avons conçu une application appelée *Analyseur* qui vérifie automatiquement la présence de motifs (*patterns*) prédéfinis dans les fichiers d'événements.

Les comportements suspects recherchés sont :
– les requêtes provoquant un arrêt total des communications (crash du serveur).
– l'absence de réponses HTTP (contient uniquement des trames TCP ou BIP).
– les codes de statuts incompatibles avec la méthode testée.
– une réponse à une requête HTTP qui ne correspond pas à la réponse prévue par la norme.
– la génération d'erreurs APDU (SW : 6F00).

L'*Analyseur* utilise en entrée un fichier qui comporte un ensemble de motifs recherchés dans les blocs requête/réponse sauvegardés dans les fichiers d'événements. L'ensemble des fichiers d'événements est parcouru pour détecter la présence d'un des motifs prédéfinis. Parmi ces motifs :

– `NO CHANNEL DATA` : (motif spécial dans le cas de SCWS) est une réponse BIP générée dans le cas où la carte à puce répond par une trame BIP sans aucun contenu HTTP.
– `6F 00` (motif spécial pour carte à puce) : est un code de retour APDU de la carte qui signifie que la requête a levé une exception Java non traitée.
– `5XX` : cette catégorie de codes de retour HTTP correspond à une erreur interne au serveur.
– *Des associations de motifs* : pour vérifier la présence de codes de retour qui ne correspondent pas à un type de requête, nous définissons des expressions régulières composées d'une association de motifs entre la requête et la réponse. L'application « *Analyseur* » recherche dans les fichiers d'événements la présence d'une de ces expressions régulières prédéfinies. Par exemple : l'association des motifs « GET » et « 201 Created » dans un bloc requête/réponse constitue une expression régulière. En effet, selon la spécification HTTP le code de retour 201 ne peut pas être une réponse à une requête dont la méthode est GET, car ce code signifie la bonne création de la ressource envoyée alors que la méthode GET ne renvoie pas de donnée.

L'analyse automatique réduit considérablement la tâche à l'utilisateur qui peut désormais sélectionner un ensemble de failles grâce à notre application « *Analyseur* » et éventuellement parcourir les quelques événements restants manuellement pour une analyse plus approfondie si c'est nécessaire.

8.4 Conclusion

L'outil de test par *fuzzing* présenté dans ce chapitre est basé sur le logiciel Peach. Il permet de vérifier la robustesse et la conformité de l'implémentation du protocole HTTP dans un serveur Web, particulièrement les serveurs Web embarqués dans des cartes à puce. Nous supposons n'avoir aucune connaissance sur l'implémentation du protocole HTTP testé. L'avantage de notre outil est qu'il réduit considérablement le nombre de données de test en éliminant des données inutiles qui ne révéleraient aucune information pertinente. Pour cela nous avons appliqué un ensemble de mécanismes à savoir :

- conception d'une application *PyHAT* permettant de déduire l'ensemble des fonctionnalités (méthodes, versions, champs, etc.) implémentées dans l'application ciblée,
- construction d'une représentation (modèles de données, modèles d'état) des requêtes/réponses valides qui seront utilisées dans la génération des mutants,
- configuration d'un *fuzzing* parallèle.

Notre outil est particulièrement adapté à des cartes SCWS (basées sur le protocole BIP) grâce à l'interface HTTP<->BIP que nous avons développée et qui gère la communication entre le générateur de tests (au format HTTP) et le serveur SCWS (communication BIP).

Chapitre 9

Implémentation et résultats expérimentaux de *Smart-Fuzz* et *PyHAT*

Sommaire

9.1 Implémentation de *PyHAT* **133**
9.2 Résultats de l'analyse **136**
9.3 L'outil *Smart-Fuzz* **136**
9.3.1 Les modèles de données 136
9.3.2 Les modèles d'états 138
9.3.3 Les mutations 139
9.3.4 Configuration des tests 139
9.3.5 Configuration du processus d'exécution 141
9.4 Expérimentations et Résultats **141**
9.4.1 Temps d'exécution 141
9.4.2 Failles détectées 142
9.5 Conclusion **143**

Notre outil de *fuzzing* composé des applications *PyHAT* et *Smart-Fuzz* a été développé en langage *Python*. *Smart-Fuzz* se base sur le logiciel Peach qui est un outil de *fuzzing* libre d'utilisation et de modification écrit en *Python* et destiné à générer et muter des données sur un protocole quelconque. En effet, il n'est pas restreint à un modèle de données particulier. Nous avons donc défini la structure du protocole HTTP que nous voudrions tester. Dans ce chapitre, nous présentons notre implémentation des applications *PyHAT* et *Smart-Fuzz* et quelques résultats expérimentaux.

9.1 Implémentation de *PyHAT*

L'application *PyHAT* permet de dévoiler l'implémentation du protocole HTTP dans un serveur Web. Elle permet de détecter les versions, les méthodes, les encodages et les champs HTTP implémentés dans le serveur Web. Cependant, elle est aussi destinée à être utilisée sur n'importe quel

serveur Web. Au lancement de son exécution, l'utilisateur est invité à choisir le type du serveur qu'il souhaite tester.

Dans cette thèse nous nous intéressons particulièrement au cas des cartes à puce SCWS. Ces cartes sont spécifiées par l'adresse IP 127.0.0.1 et le numéro de port 3516 associé à la communication HTTP.

Listing 9.1– Exécution de PyHAT sur un SCWS

```
1  if choice == '1':
2      target = '127.0.0.1'
3      port = 3516
4      try:
5      http = HTTP(target, port, doc_root, 'scws')
6          if (__OS == 'Linux'):
7              interactive = raw_input('\033[0m\n * Enter in
                  interactive mode (y/n) ? ')
8          else:
9              interactive = raw_input('\n * Enter in interactive mode
                   (y/n) ? ')
10         if interactive == 'y':
11             interactive_mode(http)
12             http.close_bip_channel()
13             choice = ''
14             continue
15     except Exception, descr:
16         if (__OS == 'Linux'):
17             print '\033[31m\n%s\033[0m' % descr
18         else:
19             print '\n%s' % descr
20             choice = ''
21             continue
```

Une fois que le type du serveur est spécifié, *PyHAT* commence son analyse. Cette analyse peut être réalisée en mode interactif où l'utilisateur doit saisir en entrée l'ensemble des requêtes permettant de tester le protocole (*listing 9.1*) ou en mode non-interactif dans lequel les requêtes sont générées de manière automatique. L'analyse dans ce dernier cas est réalisée par quatre fonctions qui s'exécutent séquentiellement et dont la fonction de chacune consiste respectivement à :

- vérifier les méthodes implémentées ;
- vérifier les versions supportées ;
- vérifier les encodages supportés ;
- vérifier les entêtes implémentés.

La vérification d'une propriété est basée sur l'analyse des réponses du serveur HTTP, comparées aux réponses attendues en fonction des requêtes spécifiées en entrée. Dans la recherche des versions supportées par exemple, la spécification RFC 2616 du protocole HTTP stipule qu'une version est supportée sur un serveur si la réponse retournée par celui-ci indique la même version que

celle utilisée dans la requête. L'implémentation de cette propriété est présentée dans le code du *listing 9.2*.

Listing 9.2– Détection des versions HTTP supportées

```
1   # get_supported_versions is used to check what
2   # HTTP versions is managed by the server
3   # @return a list of supported HTTP versions
4
5   def get_supported_versions(self):
6       # Request-Line definition for
7       # several HTTP version
8       request_line_0_9    = ('HEAD %s ', 'HTTP/0.9', '\r\nHost: %s\r\
            n\r\n')
9       request_line_1_0    = ('HEAD %s ', 'HTTP/1.0', '\r\nHost: %s\r\
            n\r\n')
10      request_line_1_1    = ('HEAD %s ', 'HTTP/1.1', '\r\nHost: %s\r\
            n\r\n')
11      requests            = [request_line_0_9, request_line_1_0,
            request_line_1_1]
12      supported_versions  = {'HTTP/0.9':True, 'HTTP/1.0':True, 'HTTP
            /1.1':True}
13      res_list            = []
14
15      for request in requests:
16          ...
17
18          if(self._mode == 'scws'):
19           # Send query
20              self._bip.send(request[0] \% self._URL + request[1] +
                    request[2] \% self.\_host)
21              # Receive response
22              response = self._bip.recv()
23
24          # Parse response
25          if(response and self._catch_status_line(response) == []):
26              supported_versions[request[1]] = False
27          elif(not response or (response and self.\_catch\_status\
                _line(response)[0] != request[1])):
28              supported_versions[request[1]] = False
29          else:
30              res_list.append(request[1])
31
32       self.server_characteristics['Supported versions'] = res_list
33       return supported_versions
```

Dans cet exemple, *PyHAT* génère trois types de requêtes HTTP (request_line_0_9, request_line_1_0, request_line_1_1), chacune indiquant une version HTTP différente (HTTP/0.9, HTTP/1.0, HTTP/1.1). Dans le cas où la cible est un serveur SCWS, à la ligne 20, la fonction *self._bip.send()* se charge d'encapsuler la requête HTTP dans un format BIP avant de l'envoyer au serveur SCWS. Pour chacune de ces requêtes, la réponse HTTP retournée par le serveur est analysée. Si la version HTTP utilisée dans la réponse est la même que celle définie dans la requête alors *PyHAT* ajoute cette version à la liste des versions HTTP supportées.

9.2 Résultats de l'analyse

Les résultats de *PyHAT* sont sauvegardés dans un fichier au format XML nommé `assesment.xml`. Ce fichier utilise principalement trois balises XML que nous avons définis :

- `<ImplementedMethods>` : définit les méthodes supportées,
- `<SupportedVersions>` : définit les versions supportées,
- `<SupportedEncodings>` : définit les encodages supportés,
- `<ParsedHeaders>` : définit les entêtes implémentés.

L'utilisation de ces différentes balises XML facilite la lecture automatique du fichier `assesment.xml` par l'outil de *Smart-Fuzz*, qui limite la génération des données de test uniquement à des fonctionnalités présentes dans ce fichier.

9.3 L'outil *Smart-Fuzz*

Peach utilise des fichiers de configuration écrits en XML. Chaque fichier contient une ou plusieurs structures/blocs de données qui permettent de décrire le format des données de test pour une l'application ou le système que l'on voudrait tester. Il fournit un ensemble de balises XML telles que : `<DataModel>, <StateModel>, <String>, <Number>`, etc. permettant de représenter les modèles de données et les modèles d'états pour la description de la grammaire du protocole à tester et pour spécifier les champs à muter et leur type.

Afin d'utiliser Peach pour tester le protocole HTTP, nous avons construit un ensemble de fichiers de descripteurs des requêtes HTTP. Ces fichiers sont composés de modèles de données et des modèles d'état.

9.3.1 Les modèles de données

Les modèles de données sont un moyen de représenter la grammaire de l'application visée. Pour cela, on utilise la balise `<DataModel>`. Il est également possible de définir des valeurs par défaut d'un modèle de données, de définir le type des données insérées (`<String>, <Number>`...) et de choisir les champs qu'on souhaite muter. Chaque balise peut contenir un ou plusieurs attributs (ref, name, value...).

Exemple : La première ligne d'une requête HTTP utilisant la méthode GET est définie comme ceci :

```
RequestLine = Method SP RequestURI SP HTTPVersion CRLF
```

Elle est représentée par un modèle de données comme suit :

Listing 9.3– représentation de la première ligne d'une requête HTTP

```
1 <DataModel name="RequestLine">
2     <String name="Method" value= "GET"/>
3     <String value=" "/>
4     <String name="RequestUri"/>
5     <String value=" "/>
6     <Block name="HttpVersion" ref="HttpVersion"/>
7     <String value="\\r\\n"/>
8 </DataModel>
```

La balise `<DataModel>` est identifiée par l'attribut `name` qui indique quelle partie de la grammaire est modélisée. Cette balise contient des sous balises `<String>` décrivant les différents champs composant une ligne de requête. Chaque champ est généralement décrit par son nom et sa valeur.

La configuration de Peach offre la possibilité de référencer un bloc de données à partir d'un autre bloc. Nous avons exploité cette propriété pour découper la grammaire du protocole HTTP en blocs et réduire ainsi la quantité de code à générer, en évitant des redondances de représentation des mêmes champs présents dans des requêtes différentes. La reconstitution d'une requête HTTP est réalisée par référencement.

Dans l'exemple précédent, le champ `HttpVersion` est référencé dans une structure `<bloc>` dont l'attribut `name` indique le nom du champ et l'attribut `ref` fait référence au modèle de données identifié par le nom `HttpVersion`. Le modèle de données `HttpVersion` est défini séparément dans un autre modèle de données. Une référence ne peut être utilisée que dans une balise représentant un bloc de donnée (`<DataModel>, <Bloc>`)

Le modèle de données représentant une requête HTTP complète regroupe tous les blocs représentant les différents champs constituant cette requête (la ligne de requêtes et les entêtes). Dans l'exemple ci-dessous (code 9.4) le modèle de données d'une requête HTTP est composé d'une suite de blocs faisant références à d'autres modèles dont le premier est `RequestLine` qui représente la première ligne d'une requête.

Listing 9.4– Modèle de données d'une requête HTTP

```
1 <DataModel name="HttpRequest">
2     <Block name="RequestL" ref="RequestLine"/>
3
4 </DataModel>
```

Il est également possible de réutiliser un même bloc pour représenter un autre type de données de même format en changeant simplement les valeurs des attributs des différentes structures de données. Cette propriété réduit considérablement le temps nécessaire pour représenter toute la grammaire HTTP. En effet, pour représenter d'autres types de requêtes, il suffit de reprendre le

modèle construit pour une requête GET et de l'adapter pour d'autres types de requêtes. Dans l'exemple du modèle de données `RequestLine` *(Listing 9.5)*; pour représenter une requête POST, il suffit de changer la valeur du champ `Method` par `POST`.

Listing 9.5– Modèle de données d'une requête POST

```
1 <DataModel name="HttpRequest">
2     <Block name="RequestL" ref="RequestLine">
3         <String name="Method" value= "POST"/>
4     </Block>
5
6 </DataModel>
```

La structure `<Data>` permet de désigner une nouvelle valeur à un champ défini dans un modèles de données. Ce sont ces valeurs qui seront prises en compte durant la génération des données de *fuzzing*. Dans l'exemple *listing 9.6*, nous réutilisons le modèle de donnée `HttpRequest` pour représenter une requête GET.

Listing 9.6– La structure <Data>

```
1 <Data name="DataRequest" DataModel="HttpRequest">
2     <Field name=" RequestL.Method" value="GET"/>
3 </Data>
```

9.3.2 Les modèles d'états

Après avoir représenté la grammaire HTTP sous forme de modèles de données, il est nécessaire de créer la structure que Peach va prendre en compte pour lui spécifier les modèles de données qu'il doit générer et définir la suite des actions à effectuer. Un modèle d'état permet également d'indiquer si les données représentées dans un modèle de données vont être envoyées ou bien reçues par l'outil de *fuzzing* (output ou input). Un modèle d'états est représenté par la balise `<StateModel>` qui peut avoir une ou plusieurs sous-balises `<State>`, chacune d'elles ayant une ou plusieurs sous-balises `<Action>`.

Voici un exemple de modèle d'état :

Listing 9.7– modèle d'état

```
1 <StateModel name="State1" initialState="Initial">
2     <State name="Initial">
3         <Action type="output">
4             <DataModel ref=" HttpRequest "/>
5             <Data ref="DataRequest"/>
6         </Action>
7     </State>
```

```
8 </StateModel>
```

L'attribut `initialState` de la balise `<StateModel>` précise le nom de la première balise `<State>` à exécuter.

Dans cet exemple, on veut générer des données définies par le modèle de données « Test ». Il est utile d'avoir plusieurs `<Action>/<State>` lorsqu'on doit gérer plusieurs choses en même temps. Par exemple, pour le protocole HTTP, si on veut aussi recevoir les réponses du serveur pour pouvoir les traiter, alors il suffit de rajouter une balise `<Action>` de type `input` et de créer un modèle de données implémentant les réponses.

9.3.3 Les mutations

Afin de générer des données semi-aléatoires, nous avons utilisé les mutateurs (*mutators*) qui définissent un type des données à muter. Comme présenté précédemment, les données sont représentées dans des balises `<DataModel>` ou `<Data>`. Ces données sont utilisées par l'outil de `fuzzing` pour effectuer des mutations et générer les données de test.

La mutation d'un champ consiste à affecter des valeurs aléatoires à ce champ mais en se limitant à un ensemble de caractères prédéfinis. Peach propose un ensemble de mutateurs de types différents tels que `StringMutator` qui agit sur les données de type `String`. L'ensemble des mutateurs supportés est contenu dans le fichier `default.xml` présent à la racine de Peach. Cependant, il est possible de définir ces propres mutateurs.

Dans certains cas, la valeur affectée à un champ peut être limitée à un ensemble de valeurs bien définies, ceci est le cas par exemple des champs spécifiant l'encodage, la version HTTP, la méthode HTTP, etc. Dans ce cas, il est plus judicieux de limiter les mutations à un ensemble prédéfini de valeurs possibles. La configuration, dans ce cas, est implémentée dans une balise `<choice>`. L'exemple du *listing 9.8* montre l'utilisation de cette balise pour la définition des valeurs possibles du champ `Method`.

Listing 9.8– Mutations sur un ensemble de valeurs bien défini

```
1 <choice>
2     <String value="GET" mutable="false" />
3     <String value="POST" mutable="false" />
4     <String value="HEAD" mutable="false" />
5     <String value="PUT" mutable="false" />
6     <String value="DELETE" mutable="false" />
7     <String value="CONNECT" mutable="false" />
8     <String value="OPTIONS" mutable="false" />
9 <choice>
```

9.3.4 Configuration des tests

La configuration d'un test consiste à définir pour ce test : le modèle d'état, la cible du *fuzzing* et d'autres options de configuration telles que les types de mutateurs à utiliser et les éléments

qui ne nécessitent pas d'être mutés. Cette configuration est implémentée dans des balises `<Test>` composées d'un ensemble de sous balises dont `<StateModel>` et `<publisher>` sont obligatoires.

Dans l'exemple du *listing 9.9*, nous faisons le choix de ne pas muter le champ `RequestL` en le précisant par la balise `<Exclude>`. Pour les autres champs que nous voudrions faire muter, deux types de mutateurs sont appliqués, à savoir : `StringCaseMutator` et `StringMutatorWithValidData`. La balise `<Publisher>` permet de spécifier la cible. Elle est composée d'un ensemble de paramètres pour spécifier certaines informations utiles. La cible dans cet exemple est un serveur standard www.unilim.fr qui communique sur le réseau internet suivant le protocole TCP et utilise le port 80 pour les communications HTTP.

Listing 9.9– Configuration des tests

```
1  <Test name="MutableTest ">
2      <Exclude xpath="//RequestL"/>
3
4      <Mutator class="string.StringCaseMutator" />
5      <Mutator class="string.StringMutatorWithValidData" />
6
7      <StateModel ref="State1"/>
8
9      <Publisher class="tcp.Tcp">
10         <Param name="host" value="www.unilim.fr" />
11         <Param name="port" value="80" />
12     </Publisher>
13 </Test>
```

Dans le cas des serveurs SCWS (basé sur le protocole BIP), la cible spécifiée est le programme PCSC qui joue le rôle d'une interface HTTP<->BIP. Il se charge de transmettre les données dans leur format convenu, de recevoir les réponses du serveur et de sélectionner les parties des données à sauvegarder dans les fichiers d'événements.

Listing 9.10– Configuration de la cible

```
1  <Test name="TestGet ">
2      <Publisher class="PCSC.PCSC">
3          <Param name="Reader" value="0" />
4          <Param name="FolderLog" value="c:/peach/http/logs/GET" />
5      </Publisher>
6
7  </Test>
```

9.3.5 Configuration du processus d'exécution

La balise `<Run>` regroupe un ou plusieurs éléments `<Test>` à exécuter ; elle permet aussi, optionnellement, de définir le mécanisme de journalisation à utiliser pour capturer les résultats de l'exécution. Les fichiers d'événements par défaut de Peach, n'enregistrent que les horaires et des cas d'un crash du système. Ces données ne nous donnent pas beaucoup d'informations sur la qualité du protocole ciblé. Nous avons donc configuré notre propre système de journalisation que nous avons implémenté dans l'application PCSC (interface HTTP<->BIP).

L'exemple du *listing 9.11* illustre la configuration d'une balise `<Run>` :

Listing 9.11– Configuration du processus d'exécution

```
1 <Run name="DefaultRun">
2     <Test ref=" MutableTest " />
3     <Logger class="logger.Filesystem">
4         <Param name="path" value="c:\\\\peach\\logtest" />
5     </Logger>
6 </Run>
```

9.4 Expérimentations et Résultats

Pour vérifier la performance et l'efficacité de notre outil, nous l'avons appliqué sur des cartes à puce MNFC, basées sur la plateforme Java Card 2.2 et qui embarquent un serveur Web SCWS. Nous avons également effectué des tests sur un prototype de carte Java Card 3. Les résultats de nos tests ont démontré l'efficacité de notre outil à détecter des failles dans le protocole testé et à mesurer l'apport des différentes optimisations apportées par notre outil.

9.4.1 Temps d'exécution

Afin de calculer la performance en temps d'exécution de notre outil, nous avons testé le *fuzzing* sur une carte à puce sans utiliser l'application *PyHAT*, le *fuzzing* dans ce cas a nécessité plus de 8 jours (nous avons arrêté l'exécution du *fuzzing* avant sa fin). L'application *PyHAT* réduit nettement ces métriques, en limitant le *fuzzing* à des méthodes, des entêtes, des encodages ou des versions implémentés dans la carte à puce testée. Le temps d'exécution a été réduit en moyenne à 6 jours.

Bien que *PyHAT* ait considérablement amélioré la performance du *fuzzing*, le temps d'exécution reste important, notamment dans le cas de test sur des cartes à puce. Nous avons donc appliqué un *fuzzing* parallèle afin de réduire la charge des données imposées à une carte. Le temps d'exécution est relatif au nombre de cartes utilisées. Nous avons appliqué notre outil sur trois cartes en parallèle. Cependant, une des cartes ne terminait pas son exécution, sans doute parce que des mécanismes de sécurité étaient implémentés sur cette dernière. N'ayant aucune connaissance sur l'implémentation des cartes que nous avons utilisées, nous n'avons pas réussi à déceler la cause exacte de ce cas. Le *fuzzing* appliqué sur les deux cartes à puce restantes a réduit le temps d'exécution généré par l'utilisation d'une seule carte de presque la moitié. Ce temps a été d'environ 56h.

9.4.2 Failles détectées

Les cartes à puce que nous avons testées nous ont révélé quelques vulnérabilités et des cas de non-conformité à la spécification, parmi ces cas :

Autorisation d'ajout ou de suppression d'une ressource : nous avons remarqué que certaines cartes acceptent les commandes PUT et DELETE et les commandes d'administration. La commande PUT permet de créer ou de modifier une ressource et la commande DELETE permet de supprimer une ressource. Ces commandes d'administration sont destinées à être utilisées par l'émetteur de la carte (administrateur de l'application) qui accède de manière sécurisée au serveur de la carte à puce pour effectuer des tâches telles que l'ajout ou la suppression d'un utilisateur, modification d'un fichier de configuration etc. L'ensemble de ces tâches ne devrait pas être accessible par l'utilisateur qui pourrait effectuer des modifications malintentionnées dans le but d'augmenter ses privilèges (par exemple : augmenter le nombre de points sur une application de fidélité) ou attaquer la carte d'une tiece personne. Certaines cartes à puce testées ont révélé des failles dans cette propriété. En effet, l'envoi de la requête PUT a permis d'ajouter une ressource HTML dans la carte. Afin de vérifier que la modification a bien été prise en compte, nous avons exécuté une requête GET qui demande de retourner la page HTML ajoutée. Nous avons constaté que la réponse de la carte contenait bien dans son corps, la page HTML que nous avions ajoutée. La fonction DELETE est aussi autorisée, nous avons réussi à supprimer cette ressource de la carte.

Génération d'une exception Java : une erreur au niveau de la carte peut engendrer un code d'erreur de type 6F XX qui correspond à une exception Java. Pour des raisons de sécurité, les cartes à puce masquent les détails de l'erreur en retournant dans tous les cas d'exception Java, le code 6F 00. Les résultats obtenus ont révélé que les cartes à puce SCWS testées génèrent une exception Java (6F 00) suite à l'injection d'une requête vide.

Réponse HTTP non conforme à la spécification : comme présenté précédemment, notre outil permet de détecter des non conformités avec la spécification, lorsque les réponses du serveur à des requêtes bien spécifiées ne correspondent pas à ce qui est prévu dans la spécification. Parmi les résultats de notre analyse nous avons constaté que lors de l'envoi d'un entête `If-Match` invalide, les cartes repondent par le code d'erreur HTTP 500 qui signifie qu'une erreur interne au serveur s'est produite, alors que la carte doit envoyer une erreur HTTP 501 (Non implémenté) ou une erreur HTTP 400 (requête incorrecte).

Implémentation incomplète : le respect de la conformité à une spécification consiste également à vérifier que tous les champs obligatoires sont bien implémentés. Les résultats obtenus ont montré que les cartes testées n'implémentent pas correctement certaines règles définies dans la spécification :

- Le champ Host qui définit le domaine de l'application ciblé est obligatoire à partir de la version HTTP/1.1. Les tests que nous avons effectués en injectant une requête sans inclure ce champ obligatoire n'a engendré aucune erreur.
- Le dernier entête d'une requête HTTP et son corps doivent être séparées par deux retours à la ligne. Les résultats que nous avons obtenus montrent que cette règle est ignorée dans les

cartes SCWS que nous avons testé. En effet, en envoyant une requête sans respecter cette règle la carte renvoie la réponse APDU `90 00` signifiant que la transaction s'est bien déroulé mais la réponse ne contient aucune réponse HTTP (pas de réponse du serveur SCWS).

Les tests que nous avons réalisé sur des cartes à puces Java Card 3 ont révélé que seules les méthodes GET et POST étaient implémentées, alors que selon la spécification de Java Card 3, cette plateforme est basée sur le protocole HTTP/1.1, qui définit les méthodes GET, POST, HEAD comme les méthodes qui DOIVENT être obligatoirement implémentées. Cependant, ne s'agissant que du premier et unique prototype (à notre connaissance), il était très probable que l'implémentation ne soit pas complète. L'outil que nous avons proposé permet de faire un audit des serveurs Web embarquées dans les cartes à puce. Cependant, il serait intéressant dans des travaux futurs de tenter d'exploiter ces failles pour effectuer des attaques.

9.5 Conclusion

Dans ce chapitre, nous avons exploré l'efficacité de la méthode de *fuzzing* pour découvrir des vulnérabilités ou des erreurs de mise en œuvre du protocole HTTP embarqué dans une carte à puce. Nous avons utilisé le logiciel Peach pour développer notre outil qui génère des données de test de manière intelligente grâce à l'utilisation de nos propres modèles de données et des modèles d'état. Notre outil est dédié à tout type de serveurs Web, y compris les serveurs Web embarqués basés sur SCWS ou la plateforme Java Card 3. Nous utilisons l'approche en boîte noire. Pour cela, notre application *PyHAT* est utilisée pour découvrir toutes les fonctionnalités mises en œuvre sur la cible, afin de limiter, par la suite, notre analyse aux seules propriétés existantes, et rendre ainsi notre *fuzzing* plus performant avec un temps d'exécution réduit.

Cependant, les cartes à puce ont une faible bande passante, pour améliorer encore plus la performance de notre analyse, nous avons appliqué un *fuzzing* parallèle sur de multiples cartes à puce. Les résultats expérimentaux ont démontré l'efficacité de nos propositions dans la réduction du temps d'exécution. L'analyse automatique des résultats du *fuzzing* qui est basée sur la comparaison entre les réponses du serveur et de la carte avec un ensemble de motifs bien définis a permis de détecter des vulnérabilités et des non conformités à la spécification RFC 2616 du protocole HTTP implémenté dans des cartes à puce. L'analyse sauvegarde l'ensemble des séquences complètes de requêtes qui ont conduit à une vulnérabilité.

Quatrième partie

Conclusions et perspectives

Chapitre 10

Conclusions et perspectives

Sommaire

10.1 Problématique **148**
10.2 Principales contributions **148**
10.3 Perspectives **149**
10.3.1 Analyse des applications Web 149
10.3.2 Fuzzing HTTP 151
10.4 Synthèse générale **152**

Les cartes à puce à serveur Web embarqué constituent une grande évolution. Grâce à cette nouvelle technologie, la carte devient accessible à une multitude de développeurs via un modèle de programmation basé sur des servlets. Ainsi, les délais de déploiement de nouveaux services sont sensiblement réduits. D'autre part, les services fournis par ces dispositifs sont beaucoup plus faciles d'usage à travers des interfaces graphiques ergonomiques, accessibles à l'utilisateur à partir du navigateur Web installé sur le terminal (téléphone par exemple). Les cartes à serveur Web embarqué offrent également la possibilité d'administrer à distance les applications installées sur la carte de manière sécurisée et une connexion à des applications distantes via une authentification robuste fournie par la carte.

Malgré tous ces avantages, cette nouvelle technologie n'a toujours pas connu l'évolution attendue, les spécifications étant sorties depuis 2008. En effet, d'une part l'utilisation des plateformes Java Card 3 nécessite des dispositifs haut de gamme, de performances beaucoup plus élevées que la plupart des cartes utilisées dans le monde ; et d'autre part, la plateforme SCWS dédiée à des cartes classiques, ne supporte pas les protocoles TCP/IP. Elle utilise d'autres protocoles plus adaptés tel que BIP. Il est donc nécessaire que le matériel auquel la carte sera connectée soit doté d'un protocole spécifique pour assurer la compatibilité entre les deux parties. Cependant, nous pensons qu'avec la nouvelle technologie des cartes NFC, les cartes à serveur Web embarqué vont finir par susciter l'intérêt des industriels. De plus, la vitesse à laquelle la technologie évolue laisse à croire que les cartes haut de gamme ne vont pas tarder à remplacer les cartes classiques.

10.1 Problématique

Dans le cadre de cette thèse, nous nous sommes intéressés à la sécurité de ces nouveaux dispositifs. L'ouverture de la carte au standards du Web oblige à reconsidérer toute la sécurité des cartes à puce. En effet, de nouveaux protocoles (par exemples : TCP/IP, BIP et HTTP) doivent être nécessairement implémentés dans la carte pour assurer sa communication avec le navigateur Web et les différents nœuds du réseau. Des failles d'implémentation de ces protocoles peuvent engendrer des vulnérabilités qui pourraient mettre en péril la sécurité de la carte à puce.

D'autre part, les vulnérabilités peuvent être causées par des failles d'implémentation des applications Web chargées dans ces cartes. Ces applications fournissent des interfaces qui facilitent l'interaction avec l'utilisateur. Avec les nouvelles technologies de génération de contenus dynamiques (JavaScript, AJAX, etc.), nous pouvons considérer que l'utilisateur contribue à la création de contenus dans des applications Web. En effet, il ne se limite pas à consulter des informations mais il publie aussi du contenu (texte, vidéo, musique). Les applications Web fournissent des points d'accès à partir desquels l'utilisateur est invité à entrer des données. Une personne malintentionnée peut donc exploiter ces points d'entrée pour injecter des données malicieuses dans l'application. Les attaques Web classiques exploitant les technologies du Web n'arrêtent pas de se multiplier et les cartes à puce à serveur Web embarqué n'en sont pas à l'abri.

10.2 Principales contributions

Dans ce rapport de thèse nous avons présenté les différentes spécifications existantes des serveurs Web embarqués (plateforme Java Card 3 et la plateforme basée sur les spécifications SCWS) et les risques que peuvent engendrer les nouvelles entités qui sont incluses dans ces spécifications. Notre contribution se divise en deux parties. La première concerne la sécurité des applications Web contre les attaques XSS et la seconde s'intéresse à la sécurité de l'implémentation du protocole HTTP dans les cartes à puce.

Notre première contribution durant cette thèse concerne la prévention et la détection des vulnérabilités XSS. Pour prévenir ces vulnérabilités, nous avons proposé une API JCXSSFilter de filtrage des données qui est compatible avec la Java Card 3. Cette API est composée d'un ensemble de fonctions de filtrage, chacune étant destinée à être utilisée dans un point d'insertion particulier (entité HTML, code JavaScript, contenu CSS, paramètre d'URL).

La détection des vulnérabilités XSS consiste à vérifier que l'API de filtrage est utilisée dans tout point d'insertion de données non fiables (issues de données externes à l'application) dans un programme. L'outil de détection que nous avons réalisé est basé sur une analyse statique et la technique de dépendance causale. Contrairement à la plupart des outils d'analyse statique existants, notre outil permet de détecter des attaques XSS persistantes et volatiles. Il réduit également le nombre de faux positifs grâce à une analyse interclasse sensible au contexte et une analyse fine des objets conteneurs (vecteur et tableau). Les résultats expérimentaux ont montré que notre API de filtrage permet de sécuriser les applications Web tout en étant portable sur la carte à puce et sans réduire les performances en temps de réponse de la carte. D'autre part, l'analyse statique a permis de détecter tous les cas de vulnérabilités que nous avons prévus dans nos applications de test.

La solution que nous avons adoptée pour la vérification du protocole HTTP consiste en un outil de *fuzzing* élaboré, permettant d'effectuer un test en boite noire du protocole HTTP, quel que soit le type de la plateforme sur laquelle il est installé. Notre approche augmente considérablement la performance du *fuzzing* (en temps et en pertinence de données de test), grâce à l'ensemble des optimisations que nous avons appliquées :

- une application *PyHAT* qui permet de détecter les fonctionnalités HTTP implémentées dans la cible ;
- un outil de *fuzzing Smart-Fuzz* basé sur des modèles de données ;
- configuration d'un *fuzzing* parallèle sur plusieurs cartes.

Nous avons également conçu une application d'analyse automatique qui recherche la présence de motifs prédéfinis dans les évènements enregistrés durant le *fuzzing*. Nos résultats expérimentaux ont démontré l'efficacité de la technique du *fuzzing* dans le test des cartes à puce, permettant la détection des failles de sécurité et des non-conformités à la spécification. Ce projet a été réalisé en collaboration avec des industriels qui nous ont fourni des cartes SCWS que nous avons utilisées dans nos expérimentations.

10.3 Perspectives

Les outils que nous avons proposés apportent des avantages par rapport aux solutions existantes. Cependant des améliorations peuvent être envisagées. Dans cette section nous proposons quelques travaux futurs pour chacun des outils que nous avons proposé.

10.3.1 Analyse des applications Web

Notre outil d'analyse statique est le premier outil destiné particulièrement à l'analyse des applications Web Java Card 3. Il offre l'avantage d'être précis comparé à des approches existantes, se basant sur le calcul de la dépendance causale entre l'ensemble des objets (étiquetage des objets non fiables et propagation de l'étiquette à tout objet qui rentre en relation avec un objet étiqueté) d'une application qu'ils soient d'une même classe ou de classes différentes, tout en étant sensible au contexte. Il a également l'avantage de réduire le nombre de faux positifs qui se présentent généralement en présence d'objets tableaux dans un programme. En effet, notre outil est précis sur l'élément du tableau concerné par la dépendance causale contrairement à la plupart des outils qui se limitent à traiter un tableau en tant qu'objet sans préciser l'élément concerné. Notre analyse permet à la fois de détecter des XSS volatiles et persistantes. Cependant, tel que nous l'avons expliqué dans le chapitre précèdent cette analyse a des limites. Nous allons proposer dans la suite quelques approches qui permettraient de compléter ces limites.

1. **Ordre d'analyse des méthodes de la classe HTTPServlet**

   Notre analyse s'exécute en boucle, en parcourant le graphe d'appel à partir de l'ensemble des nœuds correspondant aux fonctions de la classe `HTTPServlet` (doGet, doPost) que nous appelons « points d'entrée de l'analyse ». Elle réitère plusieurs fois en reprenant à chaque itération à partir d'un point d'entrée différent. L'ordre d'analyse de ces méthodes peut influencer sur la présence ou non d'une vulnérabilité, en particulier en présence de variables de classe ou des interfaces de partage SIO (voir chapitre expérimentations pour plus d'explications).

Afin de palier à ce problème nous avons proposé une solution exhaustive qui consiste à considérer toutes les combinaisons d'ordre d'exécution possibles et de relancer l'analyse pour chaque ordre d'exécution. Cependant, cette approche risque générer un nombre très important de faux positifs.

La seconde approche que nous proposons consiste à appliquer une « analyse supplémentaire » dans certains cas particuliers. Cette analyse applique l'algorithme suivant :

- l'analyseur « XSSDetector » parcourt l'ensemble des instructions d'une méthode,
- si l'instruction visitée est une sauvegarde persistante (putStatic) d'une donnée ou un appel à une méthode qui envoie des informations vers l'extérieur de l'application (println()), et que la donnée concernée (paramètre de la méthode, ou donnée sauvegardée en mémoire persistante) n'est pas signalée comme non fiable (ne porte pas d'étiquette) alors ;
- l'analyse supplémentaire intervient pour cette donnée. Elle consistera à vérifier si la donnée est le résultat de la manipulation d'une variable de classe. Si c'est le cas alors ;
- l'analyse supplémentaire parcourt l'ensemble de l'application pour vérifier si à un moment, une donnée non fiable est affectée à cette variable de classe ;
- si c'est le cas, alors une vulnérabilité XSS est signalée.

Plus précisément l'analyse supplémentaire proposée applique une technique de dépendance causale qui à l'inverse de celle appliquée par notre analyseur XSSDetector, commence par un état puits et remonte jusqu'à la source pour vérifier si cette dernière est fiable ou non. Le défi de cette solution est de traiter tous les cas possibles dans le calcul du chemin inverse permettant de remonter jusqu'à la source.

2. **Validation de l'API de filtrage**

   Notre API de filtrage XSSFilter, respecte parfaitement les recommandations d'OWASP. Cependant, les attaques XSS sont très malicieuses : un même script peut s'écrire de différentes façons avec différents encodages. Il serait donc nécessaire de tester notre API pour confirmer son efficacité.

   Notre proposition dans ce cadre, consiste à exploiter notre outil de *fuzzing* (développé pour analyser l'implémentation du protocole HTTP dans des cartes à puce) pour tester l'efficacité de l'API JCXSSFilter. L'application PyHAT, permet de détecter l'ensemble des pages Web présentes dans la carte, ces pages pourraient être exploitées pour recueillir l'ensemble des points d'insertion de données dans une application. L'outil de *fuzzing* va ensuite injecter un ensemble de combinaison de données de test, générées à partir de mutations sur différents ensembles d'encodages pour vérifier si un encodage peut outrepasser le filtrage de notre API. Une autre solution consiste à utiliser des outils existants comme Metasploit [Lud11] qui applique une base de données d'attaques pour tester une application Web.

3. **Vérifier si la fonction de filtrage utilisée est celle qui convient**

   L'outil d'analyse statique que nous avons proposé permet de vérifier que l'API JCXSSFilter est utilisée dans tout point d'insertion dans un programme où un danger XSS se présente. Cependant, cette API est composée de cinq fonctions, chacune destinée à filtrer un point d'insertion particulier (code HTML, attribut HTML, JavaScript, CSS, URL). Une même fonction ne peut pas être utilisée dans des points d'insertion de différents types. En effet, le code n'est pas interprété de la même façon selon le type de données. Les navigateurs Web sont dotés de différents interpréteurs et des caractères non malicieux pour les uns ne le sont pas forcément pour les autres. Malheureusement, notre outil d'analyse permet uniquement

de vérifier si une donnée est filtrée par JCXSSFilter sans contrôler laquelle des méthodes a été utilisée.

Pour améliorer notre analyse nous proposons de construire un « parseur HTML » qui parcourt le code HTML de chaque page visitée durant l'analyse. Toutes les balises HTML ou JavaScript rencontrées doivent être sauvegardées. Quand XSSDetector détecte un appel à une méthode de filtrage, le « parseur HTML » parcourt toutes les balises pré-enregistrées avant cet appel et supprime tous les couples de balises ouvrantes et leurs correspondantes balises fermantes (par exemple `<img> </img>`). La dernière balise ouvrante qui n'a aucune balise fermante correspondante définit le contexte dans lequel la méthode de filtrage a été appelée. Par exemple, si la balise est `<script>` alors la méthode de filtrage est appelée dans un contexte de JavaScript et la fonction de filtrage adaptée dans ce cas est `encodeforJavaScript()`.

La lecture du code HTML est simple, il suffit de récupérer les paramètres des méthodes qui affichent le code HTML d'une page Web tel que `HTTPSetvletResponse.getWriter().println()`. Il faut ensuite parcourir le text récupéré et sélectionner toutes les balises HTML utilisées dans ce texte. La difficulté de cette analyse réside dans la multitude des balises existantes et les différentes façons pour les représenter.

4. **Factorisation et adaptation aux cartes SCWS**

   L'API de filtrage que nous avons utilisée est destinée à des applications Java Card 3. Les API Java utilisées dans son implémentation ne conviennent pas à des cartes Java Card 2.2 à serveur SCWS. Il faudrait donc réadapter JCXSSFilter pour qu'elle soit compatible avec cette plateforme.

   D'autre part, l'outil d'analyse statique que nous avons conçu permet de détecter des failles dans des applications Java Card 3 qui sont basées sur l'API servlet 2.4. Les plateformes Java Card SCWS utilisent une autre API définie par la norme ETSI. Pour adapter notre analyse statique à la plateforme Java Card SCWS, il suffit de définir les méthodes des états source, puits et dérivations correspondant à cette plateforme.

   Afin de rendre notre outil plus simple d'usage. Nous proposons de faire une factorisation, en proposant une interface où l'utilisateur définit le type de l'application qu'il voudrait analyser, et selon son type (Java Card 3 ou plateforme SCWS), l'outil charge les fonctions correspondant aux états source, dérivation et puits, de la plateforme ciblée. Ces fonctions vont être prédéfinies dans des fichiers séparés pour chaque plateforme.

10.3.2 Fuzzing HTTP

L'outil de *fuzzing* que nous avons conçu présente beaucoup d'avantages par rapport aux outils existants notamment grâce à l'utilisation de l'application PyHAT qui rend le *fuzzing* beaucoup plus intelligent en se limitant à des fonctionnalités supportées par la cible et les modèles de données qui permet d'injecter des données ayant un format de requête HTTP valide. La complexité du *fuzzing* réside dans l'analyse des événements sauvegardés. Généralement cette analyse se fait manuellement, ce qui est très fastidieux vu la quantité d'événements générés. Notre outil propose une analyse des évènements en les comparant à un ensemble de motifs définis par un oracle. Cependant, notre base de motifs est limitée à des cas de non-conformité ou au crash de la carte. Afin d'enrichir notre oracle, il serait intéressant dans des travaux futurs d'établir un état de l'art des vulnérabilités les plus répandues sur des serveurs HTTP et de définir pour chacune des motifs qui la caractérisent.

D'autre part, les vulnérabilités détectées peuvent être exploitées pour tenter des attaques sur la carte à puce via le protocole HTTP. Parmi ces attaques nous avons les attaques Web.

L'outil *PyHAT* que nous avons développé permet de détecter les différentes pages Web présentes dans la carte à puce. Il serait donc intéressant d'exploiter cette information pour tenter des attaques telle qu'une XSS via les points d'insertion fournis par ces pages Web.

10.4 Synthèse générale

Cette thèse tourne autour de la technologie des cartes à serveur Web embarqué. Elle permet d'apporter des réponses à des questionnements sur l'intérêt d'intégrer un serveur Web dans une carte à puce et met l'accent sur les risques que peuvent apporter les nouveaux éléments intégrés dans ces dispositifs (serveur HTTP, applications Web, protocoles réseau).

Nous nous sommes intéressés à la sécurité au niveau du protocole HTTP et au niveau des applications Web. Nous avons donc proposé un outil de *fuzzing* pour tester la conformité et la robustesse du protocole HTTP, un outil d'analyse statique pour détecter des vulnérabilités par injection de code (XSS) dans des applications Web et une API de filtrage que les développeurs d'applications Web Java Card 3 peuvent utiliser pour sécuriser leur application.

Le choix des approches appliquées a été fait après un état de l'art des approches existantes. Cet état de l'art nous a permis de constater que la technique de *fuzzing* est l'approche la plus adaptée pour tester des environnements en boite noire et que la meilleure protection contre les XSS doit être apportées durant la phase de développement de l'application en prévoyant un filtrage des données non fiables.

Les expérimentations que nous avons réalisées sur nos outils ont montré leur efficacité. Cependant, des améliorations peuvent être apportées. Nous avons proposé un ensemble de travaux pour l'amélioration de l'analyse des événements dans notre outil de *fuzzing* et un ensemble de propositions pour réduire les fausses alertes dans notre outil d'analyse statique.

Cinquième partie

Annexes

Annexe A

A.1 Représentation du fichier assesment.xml

Le fichier *assesment.xml* est le résultat de l'analyse réalisée par *PyHAT* il est organisé comme suit :

```
<HTTP>
    <Characteristics >

        <ImplementedMethods>
            <Method isCaseSensitive="False"name="HEAD"/>
            <Method isCaseSensitive="False"name="TRACE"/>
            <Method isCaseSensitive="False"name="GET"/>
            <Method isCaseSensitive="False"name="CONNECT"/>
            <Method isCaseSensitive="False" name="PUT"/>
            <Method isCaseSensitive="False" name="POST"/>
            <Method isCaseSensitive="False" name="OPTIONS"/>
            <Method isCaseSensitive="False" name="DELETE"/>
        </ImplementedMethods>

        <SupportedVersions>
            <Version num="HTTP/1.0"/>
            <Version num="HTTP/1.1"/>
        </SupportedVersions>

        <ParsedHeaders>
            <Header name="Cache-Control"/>
            <Header name="Connection"/>
            <Header name="Date"/>
            . . .
            <Header name="Pragma"/>
            <Header name="Last Modified"/>
```

```
        </ParsedHeaders>

        <SupportedEncodings>
            <Encoding name=" identity "/>
        </SupportedEncodings>
    </Characteristics>
</HTTP>
```

A.2 Fichiers de journalisation de *Smart-Fuzz*

Les fichiers de journalisation de l'outil *Smart-Fuzz* sauvegardent les requêtes HTTP envoyées au serveur et les réponses reçues en conséquence. La figure suivante représente un événement sauvegardé dans ces fichiers.

```
4   |||||||||||||||||||| REQUEST 1 ||||||||||||||||||||
5
6   ... INITIALIZATION OF BIP COMMUNICATION ...
7
8   ---------- BEGIN : SEND OF THE REQUEST ----------
9   * RECEIVE DATA TERMINAL RESPONSE *
10  TRANSMIT (total): 80 14 00 00 44 01 03 01 42 01 02 02 82 81 03 01 00 36 81 32
11  TRANSMIT (request only):
12  $GET /test1.html HTTP/1.1
13  Host: 127.0.0.1:3516
14
15  $
16  RESPONSE (total):
17  STATUS WORD : 91 FE
18  ---------- END : SEND OF THE REQUEST ----------
19
20  ---------- BEGIN : RECEPTION OF THE RESPONSE ----------
21  * SEND DATA *
22  TRANSMIT (total): 80 12 00 00 FE
23  RESPONSE (total): 48 54 54 50 2F 31 2E 31 20 32 30 30 20 0D 0A 43 6F 6E 74 65
24  RESPONSE (request only):
25  $HTTP/1.1 200
26  Content-Length:137
27  ETag: "0003"
28  Content-Encoding: gzip
29  Content-Type: text/html
30
31  US‹BSNULNULNULNULNULNULNUL=1VTSTX1FFFwÁÿDLEo·¡{(x—HVTÕËDLE‡ESCGSDBELÝüÿ
32  STATUS WORD : 90 00
33
34  * SEND DATA TERMINAL RESPONSE *
35  TRANSMIT (total): 80 14 00 00 0F 01 03 01 43 01 02 02 82 81 03 01 00 37 01 FF
36  RESPONSE (total):
37  STATUS WORD : 90 00
38
39  ---------- END : RECEPTION OF THE RESPONSE ----------
```

Annexe B

B.1 Exemple de servlet

La servlet ci-dessous permet d'afficher du texte « Bonjour » à l'utilisateur lorsque celui-ci envoie une requête GET à une adresse associée par le conteneur Web à cette servlet :

Listing B.1– Exemple de Servlet

```
1  // Importer les packages necessaire a la creation de la servlet
2  import javax.servlet.*;
3  import javax.servlet.http.*;
4  import java.io.*;
5  // Etendue de la classe HttpServlet
6  public class PremiereServlet extends HttpServlet {
7
8      // D finir la m thode doGet() pour le traitement  d une
           requ te Get
9      public void doGet(HttpServletRequest req, HttpServletResponse
           res)
10     throws ServletException, IOException {
11
12     //D finir le type text/html de la response res   envoyer
13         res.setContentType("text/html");
14         //Cr ation d'un objet PrintWriter pour  l envoi  de texte
               format  au navigateur
15         PrintWriter out = res.getWriter();
16
17         // Envoyer les donnees au navigateur
18         out.println("<HTML>");
19         out.println("<HEAD><TITLE> Titre </TITLE></HEAD>");
20         out.println("<BODY><b1>Bonjour</b1></BODY>");
```

```
21            out.println("</HTML>");
22            out.close();
23        }
24    }
```

Une servlet étend la classe `HTTPServlet` qui fournit un ensemble de méthodes permettant de traiter les différents types de requêtes, par exemple `doGet()` correspond à la requête `GET`. Ces méthodes contiennent deux objets en paramètres, l'objet instance `HttpServletRequest` et l'objet instance de `HttpServletResponse`. Le premier contient des méthodes permettant de retourner des informations sur la requête. La méthode `doGet()` traite la requête et répond au client via l'objet `HttpServletResponse`. L'objet `PrintWriter` permet de retourner via des méthodes telle que `println()` des informations vers le navigateur Web.

B.2 Descripteurs de déploiement d'une application Web

Les descripteurs de déploiement sont des fichiers qui décrivent la structure, la configuration et des informations de déploiement d'une application. Il existe trois types de descripteurs pour une application Web : le descripteur de déploiement d'une application Web, *web.xml*, un descripteur spécifique à une application Java Card *javacard.xml* et le *runtime descriptor* MANIFEST.MF.

Le descripteur web.xml

web.xml est un fichier élémentaire pour le déploiement d'une application Web. Il est structuré en différents éléments :

- les paramètres d'initialisation des servlets,
- *mapping* de servlet,
- *mapping* des filtres,
- les listeners,
- configuration des sessions,
- sécurité (les rôles et la méthode d'authentification utilisée).

Qu'est ce que le *mapping* ? À la réception d'une requête par un client, le conteneur Web se charge de déterminer l'application concernée par cette requête. L'application Web sélectionnée doit avoir le plus long chemin de contexte qui correspond au début de l'URL définie dans la requête. Le container Web doit ensuite localiser les servets qui se chargeront du traitement de la requête. La localisation de la servlet se fait par l'intermédiaire du *mapping* qui associe un nom de servlet (`servlet-name`) aux différents chemins d'accès (`url-pattern`).

l'exemple ci-dessous présente un simple exemple de structure d'un fichier web.xml

Listing B.2– Exemple de structure d'un fichier web.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app
3     version="2.4"
```

```
4      xmlns="http://java.sun.com/xml/ns/j2ee"
5      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6      xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
7      http://java.sun.com/xml/ns/javacard/jcweb-app_3_0.xsd">
8      <display-name>MyApplication</display-name>
9      <servlet>
10         <servlet-name>MyServlet</servlet-name>
11         <servlet-class>SourcePackageName.MyServlet</servlet-class>
12     </servlet>
13
14     <servlet-mapping>
15         <servlet-name>MyServlet</servlet-name>
16         <url-pattern>/MyServlet</url-pattern>
17     </servlet-mapping>
18 </web-app>
```

Il faut noter que dans Java Card, contrairement à JEE, le chevauchement des valeurs `url-pattern` dans le `servlet-mapping` n'est pas supporté. Cela permet d'éviter qu'une application puisse prendre le contrôle des chemins d'accès d'une autre application.

Le descripteur MANIFEST.MF

Le fichier MANIFEST.MF est appelé *runtime descriptor*. Il permet de transmettre des informations de configuration et d'exécution de l'application Web entre le développeur, l'intégrateur et le déployeur. Il est composé d'un ensemble d'attributs liés à la sécurité et aux différents rôles d'accès. Les informations contenues dans le `runtime descriptor` incluent :

- le type de l'application (dans notre cas on s'intéresse au type Web) ;
- *mapping* des rôles relatifs aux utilisateurs autorisés et aux clients *on-card* (1) : à chaque rôle correspond une URI désignant les données d'authentification de l'utilisateur ;
- liste des interfaces partagées (SIO) auxquelles l'application a accès.

Il contient aussi des informations spécifiques à une application Web :

- chemin de contexte : contexte par défaut utilisé par le conteneur Web dans l'URI assignée à l'application Web ;
- numéro de port sécurisé : correspond au numéro de port qui devra être utilisé durant une communication HTTP sécurisée ;
- informations liées à l'authentification d'un client dans une session sécurisée.

Deux types d'utilisateurs sont définis pour la Java Card 3, appelés *Holder* et *User*. *Holder* correspond au propriétaire de la carte, il a un accès direct aux ressources. *User* désigne un l'utilisateur à distance. Il a des droits d'accès à distance à la carte. Cet utilisateur peut être par exemple un administrateur d'une application offerte par l'operateur.

Listing B.3– Exemple de structure d'un fichier MANIFEST.FM

```
1 Manifest-Version: 1.0
2 Runtime-Descriptor-Version: 3.0
```

```
3 Application-Type: web
4 Web-Context-Path: /MyApplication
5 User-Role-List: client
6 client-Mapped-To-Auth-URI: sio:///standard/auth/holder/global/admin
     /pin
```

Le descripteur Javacard.xml

Le javacard.xml est un fichier optionnel, structuré en un ensemble d'éléments comportant des informations de déploiement et de configuration spécifiques à la plateforme Java Card, à savoir :

- informations sur l'authentification ;
- liste des classes chargées dynamiquement ;
- liste des classes accessibles par l'application ;
- liste des rôles des utilisateurs et des clients *on-card*.

Listing B.4– Exemple de structure d'un fichier javacard.xml

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <javacard-app
3      version="3.0"
4      xmlns="http://java.sun.com/xml/ns/javacard"
5      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6      http://java.sun.com/xml/ns/jcns/javacard-app_3_0.xsd">
7      <!-- -->
8      <shareable-interface-classes>
9          <class name="com.sun.jchowto.sioservice.SmartShareable"/>
10     </shareable-interface-classes>
11 </javacard-app>
```

Annexe C

C.1 Notions Web

Dans cette annexe nous présentons quelques notions Web notamment les langages de balisages permettant de développer des applications dynamiques interactives, dont l'aspect est facilement modifiables. Souvent, les attaquants exploitent ces outils pour réaliser des attaques sur les application Web.

Gestion de session (les cookies). Une fois l'utilisateur identifié et authentifié, un mécanisme de suivi de session est utilisé pour éviter les demandes d'identification de l'utilisateur à chaque requête. Il existe plusieurs techniques standards de passer l'identifiant en paramètre dans la requête :

- ajout des variables d'identification à l'URL.
- passer en paramètre un champ de formulaire caché.
- utilisation de cookies.

Le moyen le plus utilisé est les cookies. Les cookies sont des « headers » stockés au niveau du client, permettant de stocker des paires clé/valeur. Les cookies concernant le domaine du site sur lequel un utilisateur se rend sont automatiquement envoyés dans les entêtes HTTP lors de chaque requête du client. La classe `javax.servlet.http.Cookie` permet de créer un objet Cookie. L'entête HTTP réservé à l'utilisation des cookies s'appelle `Set-Cookie`. L'envoi du cookie vers le navigateur We bse fait grâce à la méthode `addCookie()` de l'objet `HttpServletResponse`. Pour récupérer les cookies provenant de la requête du client, il suffit d'utiliser la méthode `getCookies()` de l'objet `HttpServletRequest`.

***HyperText Markup Language* (HTML).** HTML (en français "langage de balisage d'hypertexte") est un langage de balisage utilisé pour créer des pages Web. Il fournit un ensemble de balises qui permettent de mettre en forme un text qui peut inclure des éléments interactifs tel que des images, des vidéo, des liens. Il est interprété au niveau du navigateur qui affiche l'aspect de la page Web. En général, d'autres langages lui sont associés tels que CSS et JavaScript.

***eXtensible HyperText Markup Language* (XHTML).** XHTML est une évolution de HTML. Il se base sur la syntaxe de XML alors que on prédécesseur était basé sur le SGML.

Les balises utilisées en XHTML sont similaires à celles du HTML 4 mais leur syntaxe a été améliorée afin de rendre plus strictes les règles d'écriture du code (la fermeture des balises est obligatoire, le code est saisi en casse minuscule, les attributs des balises ne peuvent plus être laissés vides et leurs valeurs doivent être encadrées par des guillemets, ...).

***eXtensible Markup Language* (XML).** XML est un langage de balisage qui dérive du langage SGML. Il permet de faciliter l'échange de contenus entre des systèmes d'information hétérogènes. Sa syntaxe est extensible permettant de définir de nouvelles balises. Ces balises sont encadrées par des chevrons (< >).

JavaScript. JavaScript est un langage de programmation de scripts, qui est incorporé aux balises HTML, permettant d'améliorer la présentation et l'interactivité des pages Web. Ces scripts sont gérés et exécutés par le navigateur lui-même sans devoir faire appel aux ressources du serveur. Ils sont donc traités en direct et surtout sans retard par le navigateur. Il existe différentes librairies JavaScript qui regroupent un ensemble de fonctionnalités courantes souvent utilisées de manière répétitive. Elles offrent aux développeurs une programmation plus simple et plus rapide d'applications très riches et surtout compatibles avec différents types de navigateurs. Ces librairies contiennent notamment les fonctionnalités suivantes :

- le parcours des éléments DOM (donc gain de place sur la carte),
- utiliser des thèmes pour décorer les pages web,
- utiliser des widgets prêts à l'emploi (cf. DatePicker de JQuery UI).

Ces librairies sont stockées soit sur le serveur (la carte), soit sur un serveur distant http ://ajax.googleapis.com/ajax/libs, ce qui apporte un gain d'espace sur la carte. La *Google Libraires API* est un réseau de distribution de contenus. L'utilisation de la méthode `google.load()` augmente la vitesse des applications, tout en donnant accès à une liste évolutives de bibliothèques dont celle du JavaScript "open-source".

La librairie JavaScript la plus utilisée de nos jours est la JQuery. La communauté de JQuery a aussi développé un ensemble de composants pour les interfaces graphiques : JQuery UI.

***Document Object Model* (DOM).** Le DOM définit un moyen standard d'accès et de modification dynamique du contenu, de la structure et du style d'un document XML ou HTML, en utilisant un modèle orienté objet. Il permet aux programmes et scripts d'accéder ou de modifier individuellement chaque élément d'une page et de créer une partie d'une page.

Il fournit un ensemble d'objets, contenant des propriétés, des méthodes et des évènements que le développeur peut utiliser dans la manipulation d'un document XML ou HTML.

***Cascading Style Sheets* (CSS).** CSS appelé aussi feuille de style est un langage utilisé pour définir l'aspect d'une application Web, (la couleur du fond de la page ou le type de police). Il utilise un petit fichier (exemple "style.css") dans lequel le développeur peut définir l'aspect futur de l'application Web. Il permet de réduire le code HTML en taille et en complexité et de faciliter la modification de la présentation des pages Web en modifiant simplement le fichier CSS.

***Asynchronous JavaScript and XML* (AJAX).** L'architecture AJAX désigne l'organisation d'un ensemble d'élements (JavaScript, les CSS, XML, le DOM et le XMLHttpRequest) permettant de construire des applications Web dynamiques et interactives. Elle permet de modifier une partie d'une page Web affichée par un navigateur Web sans avoir à recharger la page entière.

Le terme "Asynchronous" (asynchrone en français), signifie que l'exécution de JavaScript ne s'arrête pas en attente d'une réponse du serveur, contrairement au mode synchrone dans lequel le navigateur serait bloqué en attendant la réponse.

Bibliographie

[20007] *The web application hacker's handbook : discovering and exploiting security flaws.* John Wiley & Sons, Inc., New York, NY, USA, 2007.

[Ait04] D. Aitel. An introduction to spike, the fuzzer creation kit. *immunity inc. white paper*, 2004.

[ALK+08] Vasilios Almaliotis, Alexandros Loizidis, Panagiotis Katsaros, Panagiotis Louridas, and Diomidis Spinellis. Static program analysis for java card applets. In *Proceedings of the 8th IFIP WG 8.8/11.2 international conference on Smart Card Research and Advanced Applications*, CARDIS '08, pages 17–31, Berlin, Heidelberg, 2008. Springer-Verlag.

[All08a] Open Mobile Alliance. Oma, enabler release definition for smartcard-web-server. Technical report, OMA, April 2008.

[All08b] Open Mobile Alliance. Oma, smartcard-web-server approved. Technical report, OMA, April 2008.

[All08c] Open Mobile Alliance. Oma, smartcard web server enabler architecture server approved. Technical report, OMA, April 2008.

[All08d] Open Mobile Alliance. Smartcard-web-server. Technical report, OMA, 2008. http ://www.openmobilealliance.org/technical/release_program/scws_v1_0.aspx.

[Ant] Antisamy project. https ://www.owasp.org/index.php/Category :OWASP _AntiSamy_Project.

[Apa06] Apache. http ://jakarta.apache.org/bcel/manual.html. *Apache Software Foundation*, 2006.

[BDH11a] Guillaume Barbu, Guillaume Duc, and Philippe Hoogvorst. Java card operand stack : Fault attacks, combined attacks and countermeasures. In *CARDIS*, pages 297–313, 2011.

[BDH11b] Guillaume Barbu, Guillaume Duc, and Philippe Hoogvorst. Java card operand stack : fault attacks, combined attacks and countermeasures. In *Proceedings of the 10th IFIP WG 8.8/11.2 international conference on Smart Card Research and Advanced Applications*, CARDIS'11, pages 297–313, Berlin, Heidelberg, 2011. Springer-Verlag.

[BGH+12] Bastian Braun, Patrick Gemein, Benedikt Höfling, Michael Marc Maisch, and Alexander Seidl. Angriffe auf openid und ihre strafrechtliche bewertung - cross site request forgery, phishing und clickjacking. *Datenschutz und Datensicherheit*, 36(7) :502–509, 2012.

[BGLP08] Fabrice Bouquet, Christophe Grandpierre, Bruno Legeard, and Fabien Peureux. A test generation solution to automate software testing. In *Proceedings of the 3rd international workshop on Automation of software test*, AST '08, pages 45–48, New York, NY, USA, 2008. ACM.

[BICL11] Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Combined software and hardware attacks on the java card control flow. In *CARDIS'11*, pages 283–296, 2011.

[BL12] Guillaume Bouffard and Jean-Louis Lanet. The next smart card nightmare logical attacks, combined attacks, mutant applications and other funny things. In David Naccache, editor, *Cryptography and Security : From Theory to Applications*, volume 6805 of *Lecture Notes in Computer Science*, chapter The Next Smart Card Nightmare Logical Attacks, Combined Attacks, Mutant Applications and other Funny Things, pages 405–424. Springer, Berlin / Heidelberg, 2012.

[BLIC11] M. Barreaud, J-L. Lanet, and J. Iguchi-Cartigny. Analyse de vulnérabilités sur cartes à puce à serveur web embarqué. *SAR-SSI*, 2011.

[BMV05] D. Basin, S. Mödersheim, and L. Vigano. Algebraic intruder deductions. *IEEE transaction on communication*, pages 549—-564, 2005.

[Boc03] G. Bochmann. A general transition model for protocols and communication services. *IEEE transaction on communication*, pages 643–650, 2003.

[BRlLV03] L. Burdy, A. Requet, J. l. Lanet, and La Vigie. Java applet correctness : a developer-oriented approach. In *In Proc. Formal Methods Europe*, pages 422–439. Springer, 2003.

[CDL06] Véronique Cortier, Stéphanie Delaune, and Pascal Lafourcade. A Survey of Algebraic Properties Used in Cryptographic Protocols. *Journal of Computer Security*, 14(1) :1–43, 2006.

[Che] Checkstyle 5.6. http ://checkstyle.sourceforge.net/.

[CMcMW04] Dixon Clare, Gago Mari-carmen, Fernández, Fisher Michael, and Der Hoek Wiebe, Van. Using temporal logics of knowledge in the formal verification of security protocols. pages 148–151, 2004.

[CnH02] Néstor Cataño and Marieke Huisman. Formal specification and static checking of gemplus' electronic purse using esc/java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right*, FME '02, pages 272–289, London, UK, UK, 2002. Springer-Verlag.

[CW09] Erika Chin and David Wagner. Efficient character-level taint tracking for java. In *Proceedings of the 2009 ACM workshop on Secure web services*, SWS '09, pages 3–12, New York, NY, USA, 2009. ACM.

[DJ04] R. Cok David and R. Kiniry Joseph. Esc/java2 : Uniting esc/java and jml - progress and issues in building and using esc/java2. In *In Construction and Analysis of Safe, Secure and Interoperable Smart Devices : International Workshop, CASSIS 2004*. Springer–Verlag, 2004.

[ECGN06] Kirda Engin, Kruegel Christopher, Vigna Giovanni, and Jovanovic Nenad. Noxes : A client-side solution for mitigating cross-site scripting attacks. 2006.

[ED] Vela Nava Eduardo and Lindsay David. Abusing internet explorer 8's xss filters. http ://p42.us/ie8xss/Abusing_IE8s_XSS_Filters.pdf.

[ESA] Owasp enterprise security api. http ://code.google.com/p/owasp-esapi-java/.

[ETS10a] ETSI. *ETSI TS 102 223 - Smart Cards : Card Application Toolkit (CAT)*, 2010.

[ETS10b] ETSI. *ETSI TS 102 588 - Smart Cards : Application invocation Application Programming*, 2010. http ://www.etsi.org/deliver/etsi_ts/102800_102899/102835/07.00.00_60/ts_102835v070000p.pdf.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Rfc 2616 : Hypertext transfer protocol–http/1.1, june 1999. *Status : Standards Track*, 1999. http ://www.ietf.org/rfc/rfc2616.txt.

[Fin] Findbugs project.

[FNE+07] Nentwich Florian, Jovanovic Nenad, Kirda Engin, Kruegel Christopher, and Vigna Giovanni. Cross-site scripting prevention with dynamic data tainting and static analysis. In *In Proceeding of the Network and Distributed System Security Symposium (NDSS'07*, 2007.

[Fou06] Mozilla Foundation. Javascript security : Same origin. 2006. http ://www.mozilla.org/projects/security/components/same-origin.html.

[Gar09] Pierre Gardenat. Xss : de la brise à l'ouragan. 2009.

[GDB06] Bearer independent protocol (bip). Technical report, G&D, 2006.

[GDS07] Smart card web server - merging the sim and the world wide web. Technical report, G&D, 2007.

[Gem] *Gemalto et LG Electronics lancent le premier téléphone portable commercial doté de la technologie Smart Card Web Server.* http ://www.gemalto.com/php/pr_view.php ?id=438.

[Gir07] Christophe Giraud. *Attaques d'écosystèmes embarqués et contre-mesures associées.* PhD thesis, Université de Versailles Saint-Quentin-en-Yvelines, 2007.

[Gol08] Dieter Gollmann. Securing web applications. *Inf. Secur. Tech. Rep.*, 13(1) :1–9, January 2008.

[Gri] Dan Griffin. Smart card middleware fuzz testing tool. `https://scfuzz.codeplex.com/`.

[Gui08] Hiet Guillaume. *Détection d'intrusions paramétrée par la politique de sécurité grâce au contrôle collaboratif des flux d'informations au sein du système d'exploitation et des applications : mise en oeuvre sous Linux pour les programmes Java.* PhD thesis, SUPELEC, 2008.

[GVLB08] Hiet Guillaume, Triem Tong Valérie, Viet, Mé Ludovic, and Morin Benjamin. Policy-based intrusion detection in web applications by monitoring java information flows. In *CRiSIS*, pages 53–60, 2008.

[Hol91] G. Holzmann. Design and validation of computer protocols. *Prentice Hall*, 1991.

[HP04] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12) :92–106, December 2004.

[HTL+05] Yao-Wen Huang, Chung-Hung Tsai, Tsung-Po Lin, Shih-Kun Huang, D. T. Lee, and Sy-Yen Kuo. A testing framework for web application security assessment. *Comput. Netw.*, 48(5) :739–761, August 2005.

[htm] Php labware, htmlawed.

[HTT] *Internet Engineering Task Force, HTTP over TLS.*

[HYH+04] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 40–52, New York, NY, USA, 2004. ACM.

[IBM] Findbugs, part 2 : Writing custom detectors.

[ICL10] J. Iguchi-Cartigny and J.L. Lanet. Developing a Trojan applet in a Smart Card. *Journal in Computer Virology*, 6 :343–351, November 2010.

[I.M] M.Gunes I.Maceno. The Fusil Project. `http://www.labri.fr/perso/fleury/courses/SS08/download/memoirs/adjej-gunes-maceno-memoire.pdf`.

[JB01] van den Berg Joachim and Jacobs Bart. The loop compiler for java and jml. pages 299–312. Springer, 2001.

[JBSP11] Martin Johns, Bastian Braun, Michael Schrank, and Joachim Posegga. Reliable protection against session fixation attacks. In *SAC*, pages 1531–1537, 2011.

[JC309a] Java CardTM 3.0.2 Application Programming Interface Specification, Classic Edition. SunMicrosystems, 2009.

[JC309b] Java CardTM 3.0.2 Runtime Environment (JCRE) Specification, Connected Edition. SunMicrosystems, 2009.

[JC309c] Java CardTM 3.0.2 Servlet Specification, Connected Edition. SunMicrosystems, 2009.

[JC309d] Java cardtm 3.0.2 specification, classic edition. SunMicrosystems, 2009.

[JC309e] Java CardTM 3.0.2 Virtual Machine (JCVM) Specification, Classic Edition. SunMicrosystems, 2009.

[JC309f] Java CardTM 3.0.2 Virtual Machine (JCVM) Specification, Connected Edition. SunMicrosystems, 2009.

[JL00] Lanet Jean-Louis. Are smart cards the ideal domain for applying formal methods ? In *ZB*, pages 363–373, 2000.

[JL04] Whaley John and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *In Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 131–144, 2004.

[joi09] Joint interpretation library, application of attack potential to smartcards. February 2009.

[KBBL11] Nassima Kamel, Mathieu Barreaud, Guillaume Bouffard, and Jean-Louis Lanet. Fuzzing on the http protocol implementation in mobile embedded web server. *C&ESAR 2011*, 2011.

[KGJE09] Adam Kieżun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *ICSE'09, Proceedings of the 31st International Conference on Software Engineering*, Vancouver, BC, Canada, May 20–22, 2009.

[KICL10] Nassima. Kamel, Julien. Iguchi-Cartigny, and Jean-Louis Lanet. Java card 3 web application developpement methodologie. 2010.

[KICLB10] Nassima. Kamel, Julien. Iguchi-Cartigny, Jean-Louis Lanet, and Mathieu Barreaud. Perspectives d'utilisation du serveur web embarqué dans la carte à puce java card 3. 2010.

[Kle] Amit Klein. *Cross site scripting explained.* http ://-courses.csail.mit.edu/6.857/2009/handouts/ css-explained.pdf.

[KLIC11] Nassima Kamel, Jean-Louis Lanet, and Julien Iguchi-Cartigny. Static analysis of java card web applications and prevention against javascript injection. *ICSNA*, 2011.

[KMM10] L. Kyrillidis, K. Mayes, and K. Markantonakis. Web server on a sim card. July 2010.

[Kon] Knizhnik Konstantin. jlint. http ://jlint.sourceforge.net/.

[Kse] kses - php html/xhtml filter. http ://sourceforge.net/projects/kses/.

[Lam06] Kevin Lam. Microsoft anti-cross site scripting library v1.5. 2006. http ://msdn.microsoft.com/en-us/library/aa973813.aspx.

[Lan11] J. Lancia. Un framework de fuzzing pour cartes à puce : application aux protocoles emv. *SSTiC*, 2011.

[LL05] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[Lud11] Courgnaud Ludovic. Xssf : démontrer le danger des xss. 2011.

[MF05] Sara Mota and Benjamin Fontan. Uml-based modeling and formal verification of security protocols. In *Proceedings of the 2005 ACM conference on Emerging network experiment and technology*, CoNEXT '05, pages 282–283, New York, NY, USA, 2005. ACM.

[Mica] Microsoft anti-cross site scripting library v1.5. http ://msdn.microsoft.com/en-us/library/aa973813.aspx.

[Micb] Eddington Michael. Peach fuzzing platform. `http://peachfuzzer.com/FrontPage`.

[MICL11] Barreaud Mathieu, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Analysis vulnerabilities in smart card web server. *SAR-SSI 2011 Network and Information Systems Security*, 2011.

[Mit] *MITRE.* http ://cwe.mitre.org/documents/vuln-trends/index.html.

[Mod] Modsecurity 2.7. http ://www.modsecurity.org.

[MOZ] MOZILA. Noscript 2.5.7. http ://noscript.net/.

[MP] Wojciech Mostowski and Erik Poll. Malicious code on java card smartcards : Attacks and countermeasures.

[MP07] C. Miller and Z.N.J. Peterson. Analysis of Mutation and Generation-Based Fuzzing, 2007.

[MSP11] Backesy Michael, Gerling Sebastian, and von Styp-Rekowsky Philipp. A local cross-site scripting attack against android phones. 2011.

[MVBL05] C. Martin Michael, Livshits V. Benjamin, and Monica S. Lam. Finding application errors and security flaws using pql : a program query language. In *OOPSLA*, pages 365–383, 2005.

[Nav10] Eduardo Alberto Vela Nava. *Web Application Obfuscation.* 2010.

[NCE06] Jovanovic Nenad, Kruegel Christopher, and Kirda Engin. Pixy : A static analysis tool for detecting web application vulnerabilities (short paper). In *IN 2006 IEEE SYMPOSIUM ON SECURITY AND PRIVACY*, pages 258–263, 2006.

[NN11] Pierre Neron and Quang-Huy Nguyen. A formal security model of a smart card web server. In *Proceedings of the 10th IFIP WG 8.8/11.2 international conference on Smart Card Research and Advanced Applications*, CARDIS'11, pages 34–49, Berlin, Heidelberg, 2011. Springer-Verlag.

[OMYS04] Ismail Omar, Etoh Masashi, Kadobayashi Youki, and Yamaguchi Suguru. A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability. *Advanced Information Networking and Applications, International Conference on*, 1 :145, 2004.

[OWAa] OWASP. *OWASP TOP 10 THE TEN MOST CRITICAL WEB APPLICATION SECURITY VULNERABILITIES.* http ://www.owasp.org/images/e/e8/OWASP_Top_10_2007.pdf.

[OWAb] OWASP. *XSS (Cross Site Scripting) Prevention Cheat Sheet.* https ://www.owasp.org/index.php/XSS___Prevention_Cheat_Sheet.

[OWAc] OWASP. *XSS Filter Evasion Cheat Sheet.* https ://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet.

[OWA07] Owasp top 10 for java ee. 2007.

[P.A07] A.Portnoy P.Amini. Sulley : Fuzzing framework, 2007. `http://www.fuzzing.org/wp-content/SulleyManual.pdf`.

[PHP] Phpids web application security 2.0. Available from : http ://phpids.org/faq/.

[PJM+00] Bieber Pierre, Cazin Jacques, A. El Marouani, Girard Pierre, Lanet Jean-Louis, Wiels Virginie, and Zanon Guy. The pacap prototype : A tool for detecting java card illegal flow. In *Java Card Workshop*, pages 25–37, 2000.

[PMD] Qj pro project. http ://pmd.sourceforge.net.

[QJP] Qj pro project. http ://qjpro.sourceforge.net.

[Qui11] D. Quisquater, J.J.and Samyde. Electromagnetic attack. pages 382–385, 2011.

[RAF04] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. A comparison of bug finding tools for java. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, ISSRE '04, pages 245–256, Washington, DC, USA, 2004. IEEE Computer Society.

[RBL12] Tiana Razafindralambo, Guillaume Bouffard, and Jean-Louis Lanet. A friendly framework for hidding fault enabled virus for java based smartcard. In *DBSec*, pages 122–128, 2012.

[RO04] Christian Rechberger and Elisabeth Oswald. Practical template attacks. In *Information Security Applications, 5th International Workshop, WISA 2004, Jeju Island, Korea, August 23-25, 2004, Revised Selected Papers, volume 3325 of Lecture Notes in Computer Science*, pages 443–457. Springer, 2004.

[SAP06] Open source static analysis tools for security testing of java web applications. 2006. http ://www.secologic.org/downloads/testing/061219_TestToolAnalyseV1.pdf.

[SCDC+11] Mathilde Soucarros, Cécile Canovas-Dumas, Jessy Clédière, Philippe Elbaz-Vincent, and Denis Réal. Influence of the temperature on true random number generators. In *HOST*, pages 24–27, 2011.

[Ser03] *Java^{TM} Servlet Specification, Version 2.4*, Novembre 2003. https ://jira.sakaiproject.org/secure/attachment/16135/servlet-2_4-fr-spec.pdf.

[SJCP05] Wagner Stefan, Jürjens Jan, Koller Claudia, and Trischberger Peter. Comparing bug finding tools with reviews and tests. In *IN PROC. 17TH INTERNATIONAL CONFERENCE ON TESTING OF COMMUNICATING SYSTEMS (TESTCOM'05), VOLUME 3502 OF LNCS*, pages 40–55. Springer, 2005.

[SS02] David Scott and Richard Sharp. Abstracting application-level web security. In *Proceedings of the 11th international conference on World Wide Web*, WWW '02, pages 396–407, New York, NY, USA, 2002. ACM.

[Sta] *International Organisation for Standardization. ISO7816. Rap. tech. ISO, 1987.*

[STE+82] C.A. Sunshine, D.H. Thompson, R.W. Erickson, S.L. Gerhart, and D. Schwabe. Specification and verification of communication protocols in affirm using state transition models. *IEEE Transactions on Software Engineering*, 8 :460–489, 1982.

[SWS02] Joel Scambray, David Wong, and Mike Shema. *Hacking Exposed Web Applications : Web Application Security Secrets & Solutions.* Osborne/McGraw-Hill, Berkeley, CA, USA, 2002.

[TCF05] David Thomas and Andrew Hunt Chad Fowler. *Programming Ruby : The Pragmatic Programmers' Guide.* Pragmatic Bookshelf, Raleigh, NC, 2. edition, 2005.

[TDM08] A. Takanen, J. DeMott, and C. Miller. *Fuzzing for software security testing and quality assurance.* Artech House Publishers, 2008.

[Tho10] Canno Thomas. Android data stealing vulnerability. 2010. http ://thomascannon.net/blog/2010/11/android-data-stealing-vulnerability/.

[UL07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing : A Tools Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[Wal00] Larry Wall. *Programming Perl.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 3rd edition, 2000.

[WAP] Web application vulnerability scanner / security auditor.

[WAP08] Halfond William, Orso Alex, and Manolios Pete. Wasp : Protecting web applications using positive tainting and syntax-aware evaluation. *IEEE Transactions on Software Engineering*, 34 :65–81, 2008.

[WAS] *WASS*. http ://www.webappsec.org/projects/statistics/.

[WEB] Webfuzzer.

[Won] E. Wong. http ://optimalcycling.com/other-projects/notscripts/limitations/. http ://noscript.net/.

[XSSa] clinton and obama xss battle develops. http ://-news.netcraft.com/archives/2008/04/24/clinton _and_obama_xss_battle_develops.html.

[XSSb] Xss on defense.gouv.fr. http ://www.xssed.com/mirror/66908/.

[XSSc] Xss on paypal.com. http ://www.xssed.com/news/44/PayPal_is_now_offering _a_free_URL_redirection_service/.

[YX11] X. Yang and X. Xiaoyao. Modeling and analysis of security protocols using colored petri nets. *Journal of computers*, 2011.