

UNIVERSITÉ M'HAMED BOUGARA – BOUMERDES



FACULTÉ DES SCIENCES

**THÈSE DE DOCTORAT**

Présentée par :

**Mesbah Abdelhak**

Filière : Informatique

Option : Informatique

---

Rétro-conception d'application Java Card

---

Devant le Jury :

M.BADACHE	Nadjib	Prof-USTHB	Président
M. MEZGHICHE	Mohamed	Prof-UMBB	Encadreur
M. LANET	Jean-Louis	Prof-INRIA	Co-Encadreur
M.HAMADOUCHE	M'hamed	Prof-UMBB	Examineur
M.RIAHLA	Mohamed Amine	MCA-UMBB	Examineur
M.HAMMAL	Youcef	MCA-USTHB	Examineur
M.BOUFFARD	Guillaume	ING-CHER-ANSSI	Invité

Année Universitaire : 2017/2018

Les objets de sécurité comme des Secure Element (SE) procèdent du paradigme "Secure-by-Design". Ils sont protégés par des mécanismes matériels et offrent probablement le plus haut niveau de sécurité. Néanmoins ces objets peuvent être étudiés afin de déceler s'ils ne possèdent pas des vulnérabilités pouvant mettre en danger les secrets qu'ils possèdent. Ces objets sont implémentés dans les passeports électroniques, les cartes bancaires ou bien les terminaux de télécommunication. S'assurer de l'intégrité et de la confidentialité des données sensibles est donc vital.

Dans cette thèse nous montrons comment il est possible de réaliser une analyse complète d'un tel objet en développant une méthodologie basée sur l'observation d'expérience et la rétro-ingénierie des données et du code. Nous avons été amené à développer de nombreux outils afin de faciliter ces étapes. En particulier, après une phase de caractérisation des objets, nous sommes capables de tracer le graphe des objets atteignables. Nous avons aussi développé un moteur d'inférence de type permettant de calculer un chemin menant entre deux états du système pouvant représenter une séquence d'instructions valides. L'exploitation de cette analyse a permis de déchiffrer les clés cryptographiques stockées pourtant chiffrées dans le SE. Nous avons aussi pu faire revenir le système dans un état normalement inaccessible, autorisant l'exécution de méthodes privilégiées. Enfin nous avons proposé la possibilité de rendre permanente une attaque en faute initialement transiente.

L'analyse du contenu de ce SE nous a permis de proposer des contremesures pouvant bloquer une telle rétro-ingénierie. Les contrôles d'intégrité sur des éléments clés du système d'exploitation, une bonne vérification du typage des paramètres et l'offuscation du code nous semblent des techniques abordables et nécessaires pour éviter cette attaque.

Ce travail a mis en évidence que, bien que développé avec des ingénieurs reconnus dans leur domaine, des implémentations très sécurisées des algorithmes cryptographiques ne protègent pas d'une attaque par une personne motivée. La sécurité par l'obscurité ne fait que retarder le succès de l'attaque mais ne peut jamais se substituer à une conception rigoureuse et audité d'un logiciel dédié à un objet de sécurité.

**Mots clés :** Élément sécurisé, Rétro-ingénierie des données, Java Card, Attaque logicielle, Découverte de vulnérabilités

Secure elements store and manipulate assets in a secure way. They are protected by hardware mechanisms and probably offer the highest level of security. Nevertheless, these objects can be studied to detect if they do not have vulnerabilities that could endanger the secrets they possess. These objects are implemented in e-passports, identity, bank cards or telecommunication terminals. Ensuring the integrity and confidentiality of sensitive data is therefore crucial.

In this thesis we show how it is possible to perform a complete analysis of such an object by developing a methodology based on the observation's experience and the reverse engineering of data and code. We have developed many tools to facilitate these steps. In particular, after a characterization phase of the objects, we are able to draw the graph of the reachable objects. We have also developed a type inference tool that calculates a leading path between two system states that can represent a valid instruction sequence. The exploitation of this analysis made it possible to decipher the cryptographic keys stored but encrypted in the operating system. We are also able to return the system to a normally inaccessible state, allowing the execution of privileged methods. Finally we propose the possibility of making fault permanent attack which is initially transient.

This work has highlighted that although developed with engineers recognized in their field, very secure implementations of cryptographic algorithms, do not protect an attack by a motivated person. Security by obscurity only delays the success of the attack, but it can never replace a rigorous and audited design of software dedicated to a security object.

**Keywords :** Secure element, Data reverse engineering, Java Card, Software Attack, Vulnerability discovery

## ملخص

تُطور عناصر الأمان على حسب نموذج التصميم بالأمان. هذه العناصر محمية بواسطة آليات مادية وربما توفر أعلى مستوى من الأمان. ومع ذلك، يمكن دراسة هذه العناصر للكشف عما إذا كانت لا تعاني من نقاط ضعف قد تعرض للخطر الأسرار التي تمتلكها. يتم استعمال هذه العناصر في جوازات السفر الالكترونية، أو البطاقات المصرفية أو محطات الاتصالات السلكية واللاسلكية. لذلك فإن ضمان سلامة البيانات الحساسة وسريتها يعد أمر حيوي.

في هذه الرسالة، نوضح كيف يمكن إجراء تحليل كامل لمثل هذه العناصر من خلال تطوير منهجية مبنية على ملاحظة الخبرة والهندسة العكسية للبيانات و الاوامر. لقد قمنا بتطوير العديد من الأدوات لتسهيل هذه الخطوات. على وجه الخصوص، بعد مرحلة توصيف الكائنات، يمكننا تمثيل هذه الكائنات برسم بياني للكائنات القابلة للوصول. لقد طورنا أيضا محركا للاستدلال من النوع الذي يحسب مسارا عكسيا بين حالتين لتمثيل تسلسل تعليمات صالحة. يمكننا استغلال هذا التحليل في فك رموز مفاتيح التشفير المخزنة بطريقة مشفرة في نظام التشغيل. كما تمكنا من إعادة النظام إلى حالة يتعذر الوصول إليها عادة، مما يسمح لنا بتجاوز

الصلحيات. وأخيراً اقترحنا إمكانية جعل هجوم دائم على غرار انه كان مفعول الهجوم عابراً. سمح لنا تحليل محتوى هذا العنصر الآمن باقتراح تدابير مضادة يمكن أن تعوق مثل هذه الهندسة العكسية. التحقق من نزاهة العناصر الأساسية لنظام التشغيل، وتشويش شفرة واجهة برمجة التطبيقات تبدو لنا أساليب معقولة و ضرورية لتجنب هذا الهجوم.

سمح لنا تحليل محتوى هذا العنصر الآمن باقتراح تدابير مضادة يمكن أن تعوق مثل هذه الهندسة العكسية. التحقق من نزاهة العناصر الأساسية لنظام التشغيل، وتشويش شفرة واجهة برمجة التطبيقات تبدو لنا أساليب معقولة و ضرورية لتجنب هذا الهجوم.

**الكلمات الرئيسية :** عناصر الامان، الهندسة العكسية للمعلومات، جافا كارد، هجوم البرمجيات، اكتشاف الضعف

TABLE DES MATIÈRES
--------------------

<b>Résumé</b>	<b>iv</b>
<b>Glossaire</b>	<b>xix</b>
<b>Acronymes</b>	<b>xxi</b>
<b>1 Introduction générale</b>	<b>1</b>
<b>I Contexte de l'étude</b>	<b>7</b>
<b>2 Objets de sécurité : SE</b>	<b>8</b>
2.1 Qu'est ce qu'un objet de sécurité : SE . . . . .	9
2.2 Histoire de la carte à puce . . . . .	9
2.3 Classification des cartes à puce . . . . .	10
2.4 Communication avec le monde extérieur (standardisation) . . . . .	14
2.5 Plateformes . . . . .	16
2.6 Utilisations . . . . .	19
2.7 Sécurité . . . . .	19
2.8 Conclusion . . . . .	21
<b>3 La technologie Java Card</b>	<b>22</b>
3.1 Présentation Java Card . . . . .	23
3.2 Architecture Java Card . . . . .	24
3.2.1 Système d'exploitation . . . . .	24
3.2.2 L'environnement d'exécution de Java Card (JCRE) . . . . .	25
3.2.2.1 La machine virtuelle Java Card (JCVM) . . . . .	25
3.2.3 L'API Java Card . . . . .	28

3.2.4	Les applets Java Card . . . . .	29
3.3	Modèle de sécurité Java Card . . . . .	30
3.3.1	Sécurité héritée . . . . .	30
3.3.2	Sécurité spécifique . . . . .	30
3.3.2.1	Le vérificateur de byte code (BCV) . . . . .	31
3.3.2.2	Le fichier CAP . . . . .	31
3.3.2.3	Le pare-feu . . . . .	33
3.3.2.4	Le mécanisme de transaction . . . . .	33
3.3.3	GlobalPlatform . . . . .	34
3.4	Conclusion . . . . .	36
<b>4</b>	<b>Attaques sur les systèmes embarqués sécurisés</b>	<b>38</b>
4.1	Attaques logiques . . . . .	40
4.2	Attaques physiques et matérielles . . . . .	41
4.2.1	Attaques invasives . . . . .	41
4.2.2	Attaques non-invasives . . . . .	42
4.2.2.1	Attaques par analyse de courant . . . . .	43
4.2.2.2	Attaques par analyse du temps . . . . .	43
4.2.2.3	Attaques par analyse électromagnétique . . . . .	44
4.2.3	Attaque par injection de faute . . . . .	44
4.2.3.1	Modèle de faute . . . . .	45
4.3	Attaques combinées . . . . .	46
4.4	Rétro-ingénierie . . . . .	47
4.5	Conclusion . . . . .	49
<b>II</b>	<b>État de l'art</b>	<b>50</b>
<b>5</b>	<b>Attaque sur carte à puce ouverte</b>	<b>51</b>
5.1	Mauvaise implémentation et/ou compréhension de la plateforme . . . . .	53
5.1.1	Abus du pare-feu Java Card . . . . .	53
5.1.2	Abus du mécanisme de transaction . . . . .	54
5.1.3	Attaque contre le vérifieur de byte code . . . . .	55
5.1.4	Dépassement de pile et changement de contexte de méthode . . . . .	56
5.2	Code malformé . . . . .	57
5.2.1	EMAN 1 : Abus des instructions statique . . . . .	57
5.2.2	EMAN 2 . . . . .	58
5.2.3	Dépassement de pile avec un code mal formé . . . . .	59
5.2.4	Les instructions JSR/RET, changement de flot de contrôle . . . . .	60
5.3	Attaque physiques contre la couche logicielle . . . . .	61

5.3.1	Outrepasser les vérifications . . . . .	61
5.3.2	Changement de flot de contrôle . . . . .	63
5.3.3	Désynchronisation du code . . . . .	64
5.4	Conclusion . . . . .	65
<b>6</b>	<b>Rétro-ingénierie</b>	<b>66</b>
6.1	Rétro-ingénierie de code . . . . .	67
6.1.1	Les défis de l'analyse statique d'un code binaire . . . . .	68
6.1.1.1	La séparation du code des données . . . . .	68
6.1.1.2	Les binaires manquent d'informations significatives . . . . .	68
6.1.1.3	Des branchements indirects . . . . .	68
6.1.1.4	Abuser les appels et les retours . . . . .	69
6.1.1.5	Offuscation et chiffrement . . . . .	69
6.1.2	Analyse statique . . . . .	70
6.1.2.1	Désassemblage linéaire . . . . .	70
6.1.2.2	Désassemblage récursif . . . . .	71
6.1.2.3	Désassemblage hybride . . . . .	72
6.1.3	Outils . . . . .	73
6.2	Rétro-ingénierie des données . . . . .	73
6.2.1	Les données à extraire . . . . .	74
6.2.2	Techniques et outils d'identification de clés cryptographiques . . . . .	75
6.2.2.1	Brute force avec dictionnaire . . . . .	76
6.2.2.2	Recherche de régions à haute entropie . . . . .	76
6.2.2.3	Une recherche pseudo-structurale . . . . .	77
6.2.2.4	Une recherche structurale . . . . .	78
6.2.2.5	Une recherche combinée . . . . .	79
6.3	Conclusion . . . . .	79
<b>III</b>	<b>Contribution</b>	<b>81</b>
<b>7</b>	<b>Acquisition du code</b>	<b>82</b>
7.1	Extraction de la mémoire NVM . . . . .	83
7.1.1	Confusion de type de l'API . . . . .	83
7.1.2	Caractérisation de tableau . . . . .	85
7.1.3	Forger des tableaux . . . . .	88
7.1.3.1	Extraction de la mémoire RAM . . . . .	89
7.1.4	Généralisation de la méthode à d'autres cibles . . . . .	89
7.1.4.1	Concept . . . . .	90
7.1.4.2	Probabilités de trouver une structure exploitable . . . . .	91

7.1.4.3	Exploitation . . . . .	93
7.1.5	Abus des instructions a/b/saload . . . . .	95
7.2	Refonte EMAN1 . . . . .	96
7.2.1	Leurrer le pare-feu : porte dérobée . . . . .	96
7.2.2	Principe et attaque de la Refonte D'EMAN 1 . . . . .	97
7.3	Évaluation . . . . .	98
7.4	Une méthodologie pour la rétro-ingénierie . . . . .	99
7.5	Conclusion . . . . .	101
<b>8</b>	<b>Rétro-ingénierie du run-time Java Card</b>	<b>102</b>
8.1	Rétro-ingénierie des données . . . . .	103
8.1.1	Méthodologie . . . . .	103
8.1.2	Caractérisation des modèles de métadonnées . . . . .	104
8.1.3	Visualisation graphique . . . . .	104
8.1.3.1	Les métriques . . . . .	106
8.2	Rétro-ingénierie de l'algorithme de gestion de la mémoire . . . . .	107
8.2.1	Le tas statique . . . . .	109
8.2.2	Le tas dynamique . . . . .	110
8.2.3	Compréhension de l'algorithme : la clé de la structure " <b>table racine</b> " . . . . .	112
8.3	Accès au code natif : la mémoire ROM . . . . .	114
8.3.1	Auto-forge : utiliser les données du système contre le système . . . . .	114
8.3.2	Résolution dynamique des liens . . . . .	115
8.3.3	Injection de la table racine . . . . .	117
8.3.4	Caractérisation des mémoires . . . . .	118
8.4	Évaluation de la méthodologie sur un autre SE . . . . .	119
8.5	Conclusion . . . . .	120
<b>9</b>	<b>Rétro-ingénierie du code</b>	<b>122</b>
9.1	Rétro-ingénierie de l'API . . . . .	123
9.1.1	Caractérisation des composants d'un paquet . . . . .	123
9.1.2	Rétro-ingénierie de la résolution dynamique des appels de méthodes . . . . .	124
9.1.2.1	L'invocation d'une méthode avec : <code>invokestatic</code> . . . . .	125
9.1.2.2	L'invocation d'une méthode avec : <code>invokespecial</code> . . . . .	126
9.1.2.3	L'invocation d'une méthode avec : <code>invokevirtual</code> . . . . .	127
9.2	Rétro-ingénierie des instruction inconnues . . . . .	128
9.2.1	Instructions inconnues . . . . .	129
9.2.2	Caractériser le nombre d'opérandes . . . . .	130
9.2.2.1	Concept de base et contraintes . . . . .	131
9.2.2.2	Modélisation informelle . . . . .	132
9.2.2.3	Analyse dynamique . . . . .	134

9.2.3	Caractériser la pré- et post-condition . . . . .	135
9.2.4	Inférer la sémantique des instructions inconnues . . . . .	136
9.2.4.1	Caractérisation par introspection de la pile et du tas . . . . .	137
9.2.4.2	La correspondance entre l'API de référence et l'API embarquée . . . . .	137
9.3	Automatisation de la caractérisation de l'API . . . . .	138
9.4	Caractérisation des méthodes natives . . . . .	140
9.4.1	Les headers spécifiques . . . . .	140
9.4.2	Instruction spécifique : appel natif . . . . .	141
9.4.3	Caractérisation des méthodes natives . . . . .	142
9.4.4	Rétro-ingénierie du code natif . . . . .	144
9.4.4.1	Localiser le code natif . . . . .	144
9.4.4.2	Analyse statique du code natif . . . . .	145
9.5	Conclusion . . . . .	148
<b>10</b>	<b>Exploitations et contremesures</b>	<b>149</b>
10.1	Exploitations des différentes vulnérabilités . . . . .	150
10.1.1	Briser le pare-feu : contexte de sécurité . . . . .	150
10.1.2	Les méthodes de lectures/écritures natives . . . . .	151
10.1.3	Changer le non-modifiable : le cycle de vie de la carte . . . . .	151
10.1.4	Identification et déchiffrement de PIN et de clés cryptographiques . . . . .	152
10.1.4.1	Utilisation des méthodes natives . . . . .	152
10.1.4.2	Leurrer le pare-feu : Abus de l'API . . . . .	154
10.1.5	Passer le BCV : Attaque combinée persistante . . . . .	156
10.1.5.1	Caractériser la pile . . . . .	157
10.1.5.2	Étude du comportement du BCV vis-à-vis le code mort . . . . .	158
10.1.5.3	Passer le BCV et récupérer l'adresse de retour . . . . .	158
10.1.5.4	Activation du calcul de la référence du <code>Tabpackage</code> . . . . .	159
10.1.5.5	Simulation et résultat . . . . .	160
10.2	Mécanismes de défense . . . . .	161
10.2.1	Vérification des méthodes de l'API . . . . .	161
10.2.2	Vérification de l'index de l'instruction <code>getstatic</code> . . . . .	161
10.2.3	Interdire l'accès aux objets inaccessible . . . . .	162
10.2.4	Vérification de l'intégrité des objets systèmes . . . . .	162
10.2.5	Offuscation du code de l'API . . . . .	163
10.2.6	Réduire le contexte d'exécution des appels natifs . . . . .	163
10.2.7	Désapprouver l'utilisation de la méthode <code>getKey()</code> . . . . .	163
10.3	Conclusion . . . . .	164
	<b>Conclusion</b>	<b>165</b>

<b>Publications</b>	<b>169</b>
<b>Bibliographie</b>	<b>170</b>
<b>Annexe</b>	<b>179</b>