Accessing Secure Information using Export file Fraudulence

Guillaume Bouffard, Tom Khefif and Jean-Louis Lanet
Smart Secure Devices Team – University of Limoges
123 Avenue Albert Thomas, 87060 Limoges, France
Email: guillaume.bouffard@xlim.fr
Email: tom.khefif@etu.unilim.fr
Email: jean-louis.lanet@unilim.fr

Ismael Kane, Sergio Casanova Salvia
Applus – LGAI Technological Center,
Campus UAB
Apto Correos 18, 08193 Bellaterra, Barcelona, Spain
Email: ismael.kane@applus.com
Email: sergio.casanova@applus.com

***Abstract*—Java Card specification allows to load applications after the post-issuance. Each application to be installed into the card is verified by a Byte Code Verifier which ensures that the application is in compliance with the Java security rules.**

The Java Card linking process is divided in to two steps. The first one is done *off-card* by the Java Card toolchain. The second one is realized during the application installation to resolve each token by an internal reference.

In this paper, we focus on the *off-card* linker, especially the conversion part between a Java-Class item and a Java Card-Cap token. For that, we provide malicious export files which will be used by the converter. This malicious API provides the same behavior as the original one for the user. With this attack, we are able to confuse the Java Card linker.

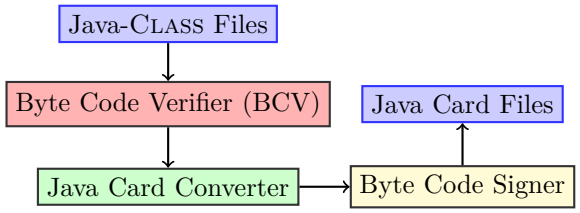
Keywords*—*Java Card, Linker, Export file, Confusion

I. Introduction

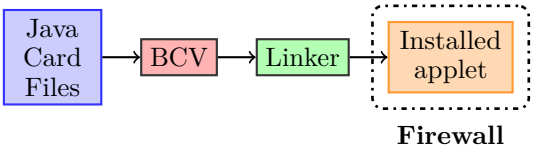
Java Card is a kind of smart card that implements one of the two editions, "*Classic Edition*" or "*Connected Edition*", of the Java Card 3.0 specification [1]. These sorts of smart cards embed a Virtual Machine (VM) which interprets codes already *romized* with the operating system or downloaded after issuance[1]. Java Card is an open platform for smart cards where a user is able to load and execute new applications after issuance. Thus different applications from different providers can run on the same card. The byte codes delivered by the Java Card toolchain are compliance with the Java Card security rules. Indeed, the loaded application is not hostile to another application into card. Furthermore, the Java Card firewall checks application permissions and access in the card, enforcing isolation between them.

A. Java Card Security Model

The Java Card platform is a multi-application environment where applet's sensitive data must be protected against malicious behavior from another applet. To protect applets properly, classical Java technology uses the type verification, Java-Class loader and security managers to create private namespaces for each applet. In a smart card, complying with the traditional enforcement process is not possible. To obtain the same security level in this type of limited-resource device, the security verification is split into two steps: *off-card* and *on-card* as shown in the Fig. 1.



(a) *off-card* security model.



(b) *on-card* security model.

Fig. 1. Java Card Security Model.

1) Off-card Java Card model: Outside the card, Fig. 1(a), the Java-application's source code is built to the Java-Class files by the Java toolchain. To ensure that the Java-Class files to be converted are semantically compliant with the Java rules, they are analyzed by the Byte Code Verifier

[1]Due to security reasons, the ability to download code into the card is controlled by a protocol defined by Global Platform [2]. This protocol ensures that the owner of the code has the necessary authorization to perform the action.

(BCV). The BCV is the main security component in the Java sandbox model: byte code alteration containing an ill-typed applet may induce a security flaw.

The Java Card based smart card is a constraint-resource device. Due to the lack of memory, the Java-Class file cannot be executed into the card. Java is supported on a Just In Time (JIT) compilation and this model is not possible into a smart card.

The chosen solution is a two-step linking process. The first one is done *off-card* where a Java-Class is translated to a tokenized Java Card-Cap – Converted APplet – file. This operation is done by the Java Card converter that provides a lightweight file optimized for constraint devices. The next linking is done inside the card. These steps are described in the section III-A.

This file format is based on the notion of interdependent components. It is specified by Oracle [1] which consists of eleven standard components: `Header`, `Directory`, `Import`, `Applet`, `Class`, `Method`, `Static Field`, `Export`, `Constant Pool`, `Reference Location` and `Descriptor`. The `Debug` component is only used for the debug process. Moreover, the targeted Java Card VM (JCVM) may support user's `custom` components. The file format is modeled in the Fig. 2.

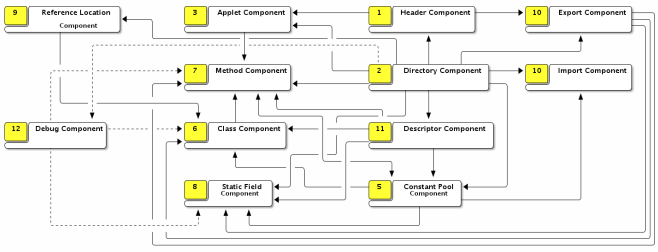


Fig. 2. The interdependent Java Card-Cap file components.

2) On-card Java Card model: Inside the card, Fig. 1(b), the Java-Cap file is checked by the embedded BCV before the installation step. After the BCV verification, the embedded Java Card linker resolves statically each token, described in the Java Card-Cap file, to a smart card internal reference. This job is a proprietary operation where each internal reference must be hided.

The separation of different applets is enforced by the firewall which is based on the package structure of Java Card and the notion of context. When an applet is created, the Java Card Runtime Environment (JCRE) uses a unique Applet IDentifier (AID) from which it is possible to retrieve the name of the package in which the applet is defined. If two applets are an instance of classes of the same Java Card package, they are considered to be in the same context. There is also a super user context, called the JCRE context. Applets associated with this context can access to objects from any other context on the card.

In this paper, we are focusing on the *off-card* linking process. Section II presents the context of our work. The Java Card linking mechanism is described in section III-A. A proof of concept to abuse the Java Card linking process is discussed in section III-B. Finally the future works and the conclusion are presented in the section IV and V.

II. Context

Java Cards have the ability to download programs after post-issuance. These applications are verified during the loading process into the card.

The idea of injecting physical fault to bypass a BCV verification has been emerged recently. A correct and legitimate application can be installed and dynamically this application can get mutate with a laser beam. This fault attack modifies a part of a memory content or a signal on an internal bus, which can lead to an exploitable deviant behavior. Some fault attacks are described in cryptanalytic papers [3], [4], [5].

G. Barbu [6] proposed a way to bypass the embedded smart card BCV. In order to do that a correct applet was installed and this applet contains an unauthorized cast between two different objects. Statically, the applet is compliant with the Java Card security rules. If a laser beam hits the bus in such a way that the cast type check instruction is not executed, this applet becomes a malware. This type of attack exploits a new method to execute illegal instructions where the physical and logical levels are perturbed. This method can work only on some cards and others are not sensitive to this attack.

Bouffard *et al.* described in [7] a way to execute a Java Card shellcode using a laser beam injection. The authors described the attacks on a loop `for` as shown in the Listing 1, but that can be extended to other instructions. The byte code version of this loop is presented in the Listing 2. The Java Card specification [1] defines two instructions to rebranch a loop, a `goto` and the `goto_w` instructions. The first one branches with a 1-byte offset and the second one takes 2-byte offset. Since the smart card's memory manager stores the array data after the memory byte code, a laser fault on the high part of the `goto_w` parameter can shift the backward jump to a forward one and they succeeded to execute the contents of an array.

```
for (short i=0 ;
     i<n ; ++i){
 foo = (byte) 0xBA;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 // Few instructions
 // have been hidden
 // for a better
 // meaning.
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;}
```

Listing 1. A `for` loop sample.

```
sconst_0
sstore_1
sload_1
sconst_1
if_scmpge_w       00 7C
aload_0
bspush            BA
putfield_b        0
aload_0
getfield_b_this   0
putfield_b        1
// Few instructions
// have been hidden
// for a better
// meaning.
aload_0
getfield_b_this   1
putfield_b        0
sinc              1 1
goto_w            FF17
```

Listing 2. Associated byte codes of the loop listed in the Listing 1.

However, the knowledge on Java Card's internal reference is needed to execute a rich shellcode. Hamadouche *et al.*, in [8], described a way to abuse the *on-card* part of the Java Card linker. As described previously, a token to resolve is preceded by a specific instruction like `invokestatic` or `getstatic`. The most of embedded Java Card does not check the correctness of the tokens referred in the `Reference Location` component. To obtain the internal references of an API, the authors created an ill-formed Java Card-CAP file where `sspush`[2] instruction takes a token as parameter. The byte code is well-formed but the Java-CAP file is ill-formed. Each token is resolved by the *on-card* Java Card linker after the *on-card* BCV checks. The most Java Card embedded BCV does not verify the instruction which precedes the tokens. This attack had been succeeded on various development cards. With this attack it is possible to know the Java Card internal references to execute a rich shellcode.

Razafindralambo *et al.* [9] abused the Java Card *on-card* linker to resolve instructions as tokens. Classically, tokens are updated after any BCV verification. For instance, if the Java Card-CAP file refers to some instructions as token, the attacker can mutate the code to a malicious byte code using the Java Card linker. To know the internal references, the attacker can use the attack described previously [8].

III. ABUSING THE LINKING PROCESS

A. Java Card Linking Process

1) Off-card linking step: Due to limited resources embedded into the card, the Java Card linking process is split into two parts. The first one is done outside the card. As described in the section I, each Java-CLASS file, complying with the Java security rules, is converted to a CAP file. To do that, the Oracle's converter translates items in the Java-CLASS file to Java-CAP file tokens . In the Java-CLASS file, an item describes the signature and the element in a string. Due to the absence of string in a Java Card, the tokenization optimizes the file size for a limited-resource device like a Java Card based smart card. To convert Java-CLASS file, each Java item is translated to a token with the help of the EXPORT files. The Oracle specification [1] specifies an EXPORT file as:

> *"An export file contains entries for externally visible items in the package. Each entry holds the item's name and its token. Some entries may include additional information as well."*

EXPORT file lists the translated names to tokens information for each method's signature, constants, classes' fields... shared by the API's owner. The EXPORT file does not contain private information, that is why it is mainly distributed publicly. For example, in the Oracle's Java Card Development Kit (JCDK), the Java Card API EXPORT files provided are compliant with each card implementing the Java Card specification [1].

[2]The `sspush` instruction push a short given in parameter.

To improve the *on-card* linking step, the Oracle's converter inserts the the Java-CAP file tokens' information into the `Constant Pool`, `Reference Location` and `Import` components. The `Constant Pool` component contains the linking information between each token's value and the reference to the method, class and/or package needs to correctly execute the byte code from the `Method` component. The `Reference Location` component lists from the `Method` component each offset where a token should be linked to a card's internal reference. The `Import` component enumerates the packages needed by the applet and listed into the `Constant Pool` component. The off-card part is described in the Fig. 1(a).

2) On-card linking step: The second linking process is done during the applet installation, after all the BCV verification. *On-card*, each token contained in the `Method` component referred by the `Reference Location` component is updated to an internal reference. The internal reference is obtained with the help of `Constant Pool` component which refers to shared methods/classes/fields needed by the application. Into the `Constant Pool` component, a token to an external method – a method provided by another class – is structured as the set (`package_token`, `class_token`, `method_token`). The package token item[3] represents a package defined in the `Import` component. So in Fig. 3, the token `2` refers to the set (`0x80`, `0x12`, `0x00`) whose method `0x00` of the class `0x12` contained in the package `0x0` indexed the `Import` component. In fact, the token `2` will be linked with the method `javacard.framework.Util.arrayCopy()` according to the EXPORT files provided in the Oracle's JCDK. With this information, the Java Card linker is able to link an applet with the needed APIs installed into the card.

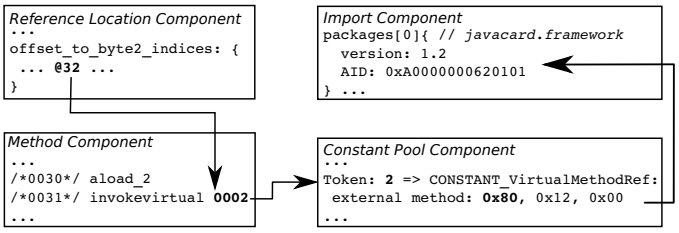


Fig. 3. *On-Card* linking.

In this paper, we focus on the *off-card* linking process in order to corrupt the compilation chain.

B. The Proof of Concept

Our objective is to perform a Man-In-The-Middle attack by forcing the *off-card* Java Card toolchain to link an application with our malicious library instead of the legitimate library required by the developer. This will allow us to withdraw confidential information like cryptographic keys. For this, a fake EXPORT file which contains the malicious linking information is inserted into the EXPORT files path. This EXPORT file is a copy of the EXPORT to confuse its AID. During the conversion step, the Java Card

[3]The Java Card specification [1] specifies that the most significant bit of the package token must be set to one.

off-card linker links each applet with the first EXPORT file which contains the carried package's name.

For instance, an API provides a function, named `buildKey`, which generates a cryptographic key. A bank applet needs a session key for cryptographic operations. For that, it must call the `buildKey` function. As this bank applet was linked with our fake API, each call to `buildKey` function is caught by our fake API. The fake API stores the generated key and return it to the caller. If our malicious library is installed into the Java Card, we can achieve fake API as described in the Fig. 4.

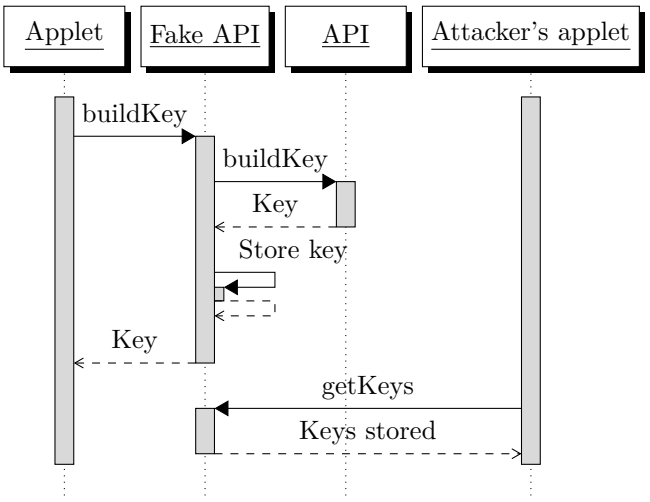


Fig. 4. The Architecture of our Man-In-The-Middle Java Card Attack.

For this attack, we speculate that the smart card loading keys are known so that it is possible to load an applet into the card.

To get a proof of concept of this attack, the following *modus operandi* is listed and the attack is presented in the Fig. 5:

1) To begin with, a copy of the API's source code to be modified is confused. The aim is to have the same classes prototype, the same methods prototype and a different package name. In fact, the Oracle's toolchain uses an alphabetic order to build a Java Card-CAP file. Keeping the same names as the original API offers us a way to friendly obtain the same EXPORT file. One or more body functions are modified to have a different behavior. For instance, an array copy can store a copy of the source array's content. This copy will be accessed later.
2) Secondly, the developer must download our fake EXPORT file and use it into his/her Java Card toolchain. This fake EXPORT file links the application from specific-API-name to our malicious API. For that, the malicious EXPORT file must contain the same names (package/classes/methods prototypes) as the original API but for its AID's which refers to the AID used in 1.
3) Thirdly, the Java-CLASS files to be converted are linked with the Java Card's converter. Since the translation from name to token is based on the EXPORT files information, the converter seeks the associated location between the name and the token. The Oracle's converter uses a *first find, first used* algorithm to find the correct EXPORT file to be used. For that, our malicious EXPORT file must be found first.
4) Finally, the applet linked with our malicious API can be installed on the card. The installed applet will use our fake API as the legitimate one and the values returned by our API should be in compliance with the original API.

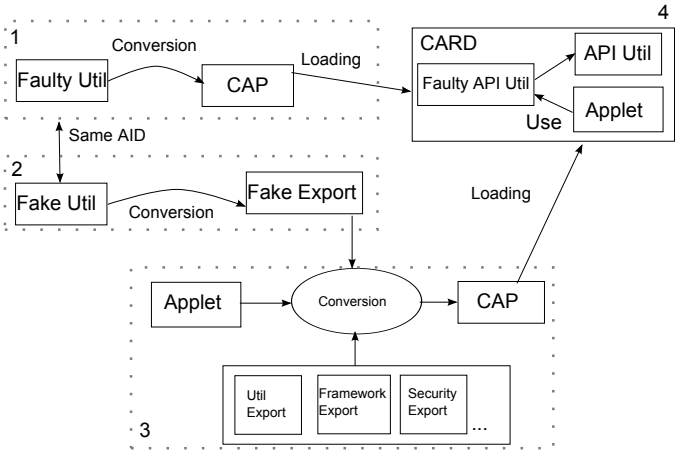


Fig. 5. Man-In-The-Middle procedure.

IV. Future Works

In Section III-B, a proof of concept for a man in the middle attack was described. It is possible to apply it on a real Java Card API provided by Oracle. The next step will be the creation of a fake `javacard.security` package which keeps a copy of each created `Key`. A malicious applet will call an undocumented API function to retrieve each created `Key`.

At this point, the Java Card firewall must check the owner of each created object. The firewall ensures a segregation between applets from different owners. When an object is used, the firewall checks only the object's ownership. In our case, the object `Key` is created by an API and shared to each API's user. The Java Card firewall cannot prevent this behavior.

Providing a malicious API can create some problems and the Java Card linker associates each type's name to a token. During the execution, the Java Card firewall dynamically checks the correctness of the object's type.

Abusing the Java Card linking process created an incompatibility between two objects' type. The incompatibility is occurred due to `javacard.security.Key` and our `javacard.security.Key'`.

In our case, the object type `javacard.security.Key` is obtained by the `javacard.security.KeyBuilder.keyBuilder()` function and cannot be casted to the type `javacard.security.Key'` declared into our malicious API. Moreover, the Java Card linker does not allow the usage of different EXPORT files

for the same package (one EXPORT file for each Java Card package).

V. Conclusion

In this paper, we presented an attack on the Java Card *off-card* linker. Actually the Java Card *off-card* linker does not check the correctness of EXPORT files used to convert a Java-Class file to a Java Card-Cap file. The Oracle's Java Card EXPORT files are supposed to be corrected without hashsum to check the files' integrity. So the developer must have trust in the provided EXPORT files.

This concept proves that Java Card *off-card* toolchain can be abused. In the next step we will apply this attack on a real API like, `javacard.security` framework. For the moment, we performed this attack only on our API without external dependencies. The applications have to respect external dependencies in order to run the applets correctly on real Java Card. Thereby using this proof of concept on `javacard.security.KeyBuilder`, the type constraints should be guaranteed to have a working result.

References

[1] Oracle, *Java Card 3 Platform, Virtual Machine Specification, Classic Edition 3.0.0*. Oracle, Sep. 2011.

[2] Global Platform, *Card Specification v2.2*. March, 2006.

[3] C. Aumüller, P. Bier, P. Hofreiter, W. Fischer, and J.-P. Seifert, "Fault attacks on RSA with CRT: Concrete Results and Practical Countermeasures," *IACR Cryptology ePrint Archive*, vol. 2002, p. 73, 2002.

[4] L. Hemme, "A Differential Fault Attack Against Early Rounds of (Triple-)DES," in *CHES*, ser. Lecture Notes in Computer Science, M. Joye and J.-J. Quisquater, Eds., vol. 3156. Springer, 2004, pp. 254–267.

[5] G. Piret and J.-J. Quisquater, "A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD," in *CHES*, ser. Lecture Notes in Computer Science, C. D. Walter, Çetin Kaya Koç, and C. Paar, Eds., vol. 2779. Springer, 2003, pp. 77–88.

[6] G. Barbu, "On the security of Java Card platforms against hardware attacks." Ph.D. dissertation, Grant-funded with Oberthur Technologies and Télécom ParisTech, 2012.

[7] G. Bouffard, J.-L. Lanet, and J. Iguchi-Cartigny, "Combined Software and Hardware Attacks on the Java Card Control Flow," in *Smart Card Research and Advanced Applications*, ser. Lecture Notes in Computer Science, E. Prouff, Ed., vol. 7079. Berlin/Heidelberg: Springer, Sep. 2011, pp. 283–296.

[8] S. Hamadouche, G. Bouffard, J.-L. Lanet, B. Dorsemaine, B. Nouhant, A. Magloire, and A. Reygnaud, "Subverting Byte Code Linker service to characterize Java Card API," in *Seventh Conference on Network and Information Systems Security (SAR-SSI)*, May 22rd to 25th 2012, pp. 75–81. [Online]. Available: https://sarssi2012.greyc.fr/

[9] T. Razafindralambo, G. Bouffard, B. N. Thampi, and J.-L. Lanet, "A dynamic syntax interpretation for java based smart card to mitigate logical attacks," in *SNDS*, ser. Communications in Computer and Information Science, S. M. Thampi, A. Y. Zomaya, T. Strufe, J. M. A. Calero, and T. Thomas, Eds., vol. 335. Trivandrum, India: Springer, 2012, pp. 185–194.