J Comput Virol (2010) 6:343–351
DOI 10.1007/s11416-009-0135-3

ORIGINAL PAPER

Developing a Trojan applets in a smart card

Julien Iguchi-Cartigny · Jean-Louis Lanet

Received: 4 January 2009 / Accepted: 26 August 2009 / Published online: 11 September 2009
© Springer-Verlag France 2009

Abstract This paper presents a method to inject a mutable Java Card applet into a smart card. This code can on demand parse the memory in order to search for a given pattern and eliminate it. One of these key features is to bypass security checks or retrieve secret data from other applets. We evaluate the countermeasures against this attack and we show how some of them can be circumvented and we propose to combine this attack with others already known.

1 Introduction

A smart card is a piece of plastic, the size of a credit card, in which a single chip microcontroller is embedded. Usually, microcontrollers for cards contain a microprocessor and different memories: Ram (for run-time data and OS stacks), Rom (in which the operating system and the "romized" applications are stored), and Eeprom (in which the persistent data are stored). Due to strong size constraints on the chip, the amount of memory is small. Most smart cards sold today have at most 5 KB of Ram, 256 KB of Rom, and 256 KB of Eeprom. This chip usually also contains some sensors (like light sensors, heat sensors, voltage sensors, etc.), which are used to disable the card when it is physically attacked.

A smart card can be viewed as a secure data container, since it securely stores data and it is used securely during short transactions. Its safety relies on the underlying hardware. A physical attack is quite difficult because the chip in a card is embedded with sensors covered with a resin, and all components are on the same chip (difficult to probe an internal bus). The software is the second barrier for its safety. The embedded programs are usually designed neither for returning nor modifying sensitive information without being sure that the operation is authorized. Java Card is a kind of smart card that implements the standard Java Card 3.0 [1] in one of the two editions "*Classic Edition*" or "*Connected Edition*". Such a smart card embeds a virtual machine which interprets codes already romized with the operating system or downloaded after issuance. 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 credentials to perform the action.

Java Card is an open platform for smart cards, i.e. able of loading and executing new applications after issuance. Thus, different applications from different providers run in the same smart card. Thanks to type verification, the bytecodes delivered by the Java compiler and the converter (in charge of delivering compact representation of class files) are safe, i.e. the application loaded is not hostile to other applications in the Java Card. Furthermore, the Java Card firewall checks permissions between applications in the card, enforcing isolation between applications. Until now, it was safe to presume that the firewall was efficient enough to avoid bad behaviour from malicious applications (crafted applet modified after off-card verification). In this paper, we will show that an attacker can generate malicious applications which bypass firewall restrictions and modify other applications, even if they don't belong to the same security package. Since the firewall is the only mandatory on-card security mechanism in the Java Card our work reveals an important security issue. Java card has been an important improvement in the smart card world due to the security aspect of the Java platform. Nevertheless some attacks have been successful in retrieving secret data from the card. Thus we will present in

J. Iguchi-Cartigny · J.-L. Lanet (✉)
XLIM/DMI/SSD, 83 rue d'Isle, 87000 Limoges, France
e-mail: jean-louis.lanet@xlim.fr

this paper a methodology to get access to data, which should bypass Java security components.

In the next section, we describe the different components involved in the Java Card's platform security. Section 3 describes the state-of-the-art of smart cards logical attacks. Then we introduce, a methodology to implement a Trojan in the card in Sect. 4. Section 5 presents the evaluation of our attack and encountered countermeasures. Finally, the last section presents our future works and concludes.

2 Java Card

Java Card is quite similar to any other Java edition, it only differs (at least for the Classic Edition) from standard Java in three aspects: (i) restriction of the language, (ii) runtime environment and (iii) the applet life cycle. Due to resource constraints the virtual machine in the Classic Edition must be split into two parts: the bytecode verifier executed off-card, is invoked by a converter while the interpreter, the API and the Java Card runtime environment (JCRE) are executed on board. The bytecode verifier is the offensive security process of the Java Card. It performs the static code verifications required by the virtual machine specification. The verifier guarantees the validity of the code being loaded in the card. The bytecode converter transforms the Java class files, which have been verified and validated, into a format that is more suitable for smart cards, the CAP file format. An on-card loader installs the classes into the card memory. The conversion and the loading steps are not executed consecutively (a lot of time can separate them). Thus, it may be possible to corrupt the CAP file, intentionally or not, during the transfer. In order to avoid it, the Global Platform Security Domain checks the integrity and authenticates the package before its registration in the card. Along this paper when talking about Java Card we will refer to the "*Classic Edition*".

2.1 The Java Card platform

Developers write Java applets which can be uploaded to Java Card smart cards. Then, the applet's bytecode is interpreted by the embedded Java Card virtual machine (JVCM), an implementation of the Java Card Platform Specification. Such smart cards are thus open platforms, where new applications (or applets) can be uploaded after issuance of the card to the final user.

Due to resource constraints, the JCVCM must be split into two parts:

- First, the bytecode verifier (BCV) and the converter are executed outside the card (off-card) to generate a valid CAP file.
- Second, the interpreter, the API and the Java Card runtime environment (JCRE) execute and handle the applet behaviour inside the smart card.

In the first part, the bytecode verifier acts as an offensive security process located in the JCVM. It performs static code analysis on the Java class files, which is required by the JVM specification. Then, the bytecode converter transforms these files into a more suitable format used with smart cards: a CAP file. This file is a JAR file containing a compact representation of one or several class files adapted to the smart card constraints.

The next step is the storage of the Java classes in the card memory by the on card loader. During the loading process, the CAP file is first unpacked into individual components, which are then downloaded into the card sequentially, component by component by the off-card loader. A special application on the card—the Installer–receives the newly downloaded applet and stores its content into the persistent memory. It also requests from the system the linking of the new classes to the packages already present on the card. After loading and linking, the package is ready to be executed by the JCVM.

2.2 Java Card security

The Java Card platform is a multi-application environment in which an applet's critical data must be protected against malicious access from other applets [3]. To enforce protection between applets, traditional Java technology uses type verification, class loaders and security managers to create private namespaces for applets. In a smart card, it is not possible to comply with the traditional enforcement process.

Firstly, the type verification is executed outside the card due to memory constraints. Secondly, class loaders and security managers are replaced by the Java Card firewall.

2.3 The bytecode verifier

Allowing code to be loaded into the card after post-issuance raises the same issues as with web applets. An applet that has not been compiled by a compiler (handmade bytecode) or that has been modified after compilation can break the Java sandbox model. Thus the client must check that the Java typing rules are preserved at the bytecode level.

The Java language is strongly typed, which means that every variable and every expression has a type that is determined at compiling time. Type mismatches in the source code are detected at compile time as well, and Java bytecode is also strongly typed. Still, local and stack variables of the virtual machine do not have unchanging fixed types even in the scope of one method execution. Not all type mismatches are detected at runtime, and this allows building malicious

applets exploiting this issue. For example, pointers are not supported by the Java programming language. Though, they are extensively used in Java Virtual Machine, where they are referred to as references. Thus, the absence of pointers reduces the number of programming errors. But it does not stop attempts to break security protections by disloyal use of pointers.

Bytecode verifier (BCV) is a crucial security component in the Java sandbox model: any bug in the verifier causing an ill-typed applet to be accepted can potentially enable a security attack. At the same time, bytecode verification is a complex process involving elaborate program analyses. Moreover such an algorithm is very costly in terms of time consumption and memory usage. For these reasons, many cards do not implement such a component and rely on the fact that it is the responsibility of the organisation that signs the code of the applet to ensure that the code is correctly typed.

2.4 Java card firewall

The separation between different applets is enforced by the firewall which is based on the package structure of Java Card and the notion of contexts. When an applet is created, the JCRE uses a unique applet identifier (AID) from which it is possible to retrieve the name of the package in which it is defined. If two applets are instances of classes coming from the same Java Card package, they are considered as belonging to the same context. There is a superuser context, called the JCRE context. Applets belonging to this context can access objects from any other context on the card.

Every object is assigned to a unique owner context which is the context of the applet that created the object. A method of an object is said to be executed in the owner context of the object. It is this context that decides whether access to another object is allowed or not. The firewall isolates the contexts in such a way that a method executing in one context cannot access any attribute or method of objects belonging to another context.

There are two ways to bypass the firewall: via JCRE entry points and via shareable objects. JCRE entry points are objects owned by the JCRE that have been specifically designated as objects that can be accessed from any context. The most significant example is the APDU buffer in which commands sent to the card are stored. This object is managed by the JCRE and, in order to allow applets to access this object, it is designated as an entry point. Other examples include the elements of the table containing the AIDs of the applets installed on the card. Entry points can be marked as temporary. References to temporary entry points cannot be stored in objects (this is enforced by the firewall).

2.5 The sharing mechanism

To support cooperative applications on a single card, the Java Card technology provides well defined sharing mechanisms. The shareable interface object (SIO) mechanism is the mechanism in the Java Card platform intended for applets collaboration. The `javacard.framework` package provides a tagging interface called Shareable, and any interface which extends the Shareable interface will be considered as a Shareable interface. Requests for services to objects implementing a Shareable interface are allowed by the firewall mechanism. When a server applet wants to provide services to other applets within the Java Card, it must define the services it wants to export in an interface tagged as Shareable.

Within the Java Card, only instances of classes are owned by applets (i.e. are within the same security context), classes themselves are not. No runtime check is performed when a static field is accessed or when a static operation is invoked. This means that static fields and operations are accessible from any applet; however, objects stored in static fields belong to the applet which instantiates them. The server applet may decide whether to publish its Shareable Interface Objects (SIOs) in static fields, or return them in static operations.

3 Related work

In order to retrieve some data from the card, we can use two different approaches: physical attacks or logical attacks.

3.1 Physical attack

A first class of physical attacks is side-channel attacks [4]. These non-invasive attacks consist in observing an unintended physical effect of computation (timing, data exchanged on the I/O channels, power consumption, electromagnetic noise, etc.) to discover information like the secret key used in some cryptographic operations. For instance, SPA and DPA stand for Simple and Differential Power Analysis, respectively, and aim at exploiting the information leaked through characteristic variations in the power consumption of electronic components. Fault induction attack, or perturbation attack, consists in changing the behaviour of the component in order to create an exploitable error [5]. Such faults can be induced using different means, including glitches (a surge of power on one of the cards I/O ports), light/laser exposure, etc. The attack will typically try to make cryptographic operations weaker (by creating faults that can be used to recover key or plaintext information), or avoid or corrupt the results of checks (such as authentication or lifecycle state checks, or else change the program flow).

3.2 Logical attack

As explained in [6] logical attacks consist in executing ill-formed applications, i.e., malicious applications that are made of illegal bytecode instructions sequences or that do not have valid bytecode parameters. Such attacks are limited to cards for which:

- post issuance is allowed,
- the attacker has the credentials (Security Domain keys),
- the card must not include a bytecode verifier.

W. Mostowski et E. Poll [7] proposed several attacks on Java Card using an ill-typed code. One way is to exploit type confusion between primitive arrays of different types. By convincing the applet to handle a byte array as a short array, it would theoretically be possible to read or write twice the size of the original byte array. They use different approaches: a flaw in the implementation of the transaction mechanism and the shareable interface.

Another way [8,9] is to trick the virtual machine to handle an object as an array. Hence, fields from a forged object can be seen as length of the array if they are stored at the same offset in the physical memory. The attacker would be able to set the size of the array and thus have access to the whole memory of the Java Card. The authors exploit the previous attack to handle reference as short (and short as reference), thus allowing reading and writing existing references, something theoretically impossible on Java Card. They proposed several exploitations of this method. First, it is possible to swap the references of two objects even if they have incompatible types. An attacker can also manipulate the system-owned applet identifier (AID) and thus impersonate a valid applet (using a stolen AID). Finally, it would be possible to spoof references and thus be able to read part of the memory. This last step was not conclusive, as most of the cards refuse spoofed references.

Hyppönen [10] proposed a way to exploit weaknesses in static instructions (*getstatic* and *putstatic*) and in the reference location component of the CAP file which can lead to reference spoofing. The *getstatic* instruction is used to get a static field from a class. The two operands of this opcode are used to build an index in the constant pool. During applet loading, the on-card linking process will replace the two operands with an address in real memory. The idea of the attack is to remove the entry in the Reference Location component so that the operand of the *getstatic* will not be resolved. Thus it becomes possible to assign any value to the two operands, and therefore to point to any real address. This attack works (in the absence of a bytecode verifier) because no context check is done during access to static field. However, the author did not present any experimental results or an implementation of the attack.

In this paper, we will prove that such an attack is possible, and we propose a very efficient implementation using another bytecode weakness that allows us to generate a safe mutable code.

4 The malicious code

Instead of dumping the memory byte after byte we use the ability to invoke an array that can be filled with any arbitrary bytecode. Within this approach, we are able to define a search and replace function. To demonstrate the application of such attack, please consider the following generic code, often used to check if a PIN code has been validated.

```
public void debit (APDU apdu )
{
...
if (!pin.isValidated())
{
ISOException.throwIt(SW_AUTH_FAILED)
}
// do safely something authorized
}
```

The pin object is an instance of class *ownerPin*, which is a secure implementation of a PIN code (ratification counter decrements before check and so on). If the user sets a wrong PIN code, an exception is thrown. The goal of our Trojan is to search for the bytecode of this exception treatment and to replace it with a predefined code fragment. For example, if the Trojan finds in memory the pattern 11 69 85 8D 00 12 and if the owner of this method is the targeted applet then the Trojan replaces it by the following pattern: 00 00 00 00 00 00. Knowing that the bytecode 00 stands for the NOP instruction, the original code becomes:

```
public void debit (APDU apdu )
{
...
if (!pin.isValidated())
{ }
// do safely something authorized
}
```

The interest of the search and replace Trojan is obvious. Of course if the Trojan is able to perform such an attack it could also scan the whole memory and characterize the object representation of the virtual machine embedded into the card. It becomes also possible to get access to the implementation of the cryptographic algorithms which in turn can be exploited to generate new attacks.

The basic hypotheses of this attack are: the attacker has the credentials to download his own code into the Java Card and the card does not include a bytecode verifier.

4.1 The mutable applet

The first step is to get a reference to an array located in the context of one of our own applets. Suppose that the following function is in our applet:

```
public short getMyAddresstabByte(byte[] tab)
{
short dummyRef=(byte)0x55AA;
tab[0] = (byte)0xFF; // second instruction
return dummyRef;
}
```

A look at the bytecode level shows that the second instruction is an aload 1, so the reference of the array tab is on the top of the stack after the second instruction. If we change each bytecode by the instruction NOP between the second instruction and the return statement, then the function will return the reference of the array tab instead of the short *dummyRef*. Thus, the array reference can be sent back to the terminal (as a short) using an APDU command. Using this bytecode manipulation of the external file we are able to get a valid reference on elements belonging to our security context.

We begin by defining an array variable (called codeD). The compiled code is the following:

```
public byte[] codeD = {(byte)0x01, (byte)0x00, (byte)0x7D,
(byte)0x00, (byte)0x00, (byte)0x78};
```

If you consider the contents of this array as a method, we have now a function dedicated to read a static field. The two first bytes correspond to the header of the method and will never be interpreted as bytecode.

```
//flags : 0
//max_stack : 1
//nargs : 0
//max_locals : 0
00 aconst_null // header of the method
01 nop // i d e m
02 getstatic_s 0 0
05 sreturn
We can modify the value of the third and fourth bytes with
an APDU command,
and thus choose the operands of getstatic s:
codeD[3]= apduBuf[ISO7816.OFFSET_CDATA];
codeD[4]= apduBuf[ISO7816.OFFSET_CDATA+1];
```

We need a dummy definition (*functionToReplace()*) to generate a reference on a static method:

```
// function to replace
static public short functionToReplace()
{
return ad;
}
```

Now, we can write the loop used to search and replace:

```
For (i=0...){
// to generate a ref to be replaced later
Util.setShort(searchBuf,k,functionToReplace());
codeDump[4]++; //increment low address
if (codeDump[4] == (byte)0x00)
{
codeDump[3]++; // increment high address
}
// search and replace the pattern in searchBuf
}
```

We use another weakness on *invokestatic* and we modify the *Method Component* in order to reference the array CodeD instead of the original method. We also have to modify the Reference Location Component of the CAP file to remove the entry as shown in the Fig. 1.

At offset 0x014e in the Method Component we find the *invokestatic* instruction calling *fonctionToReplace()*. We can see that the operand is an index referring to the *Constant Pool Component* and that the operand offset is referenced in the *Reference Location Component*. First, we edit the file and change the value of this operand offset (0x014f) by the value of its successor in the Reference Location Component. Thus the linker will resolve twice the next entry (which refers to the method *setShort()).*

In the second step, we replace the operands of *invokestatic* by the address of the array's content and then we download and install the mutable applet. Note that we need to retrieve the resolved address of this array, which leads to a two step installation. First, during the installation, we send an APDU command to retrieve the address of the array by using the function described in the Sect. 4.1. In the second step, we replace the operand of *getstatic,* denoted s, by the retrieved address, we remove the entry in the reference location component and then we download and install the mutable applet. The Trojan applet is then ready to be initialized by APDU commands at any time during the card live. If the assets to be obtained are present into the card, initialisation command and search and replace command can be process.

Then the invokestatic will invoke the buffer code and we will just have to increment the adequate location in the buffer to parse the whole memory. As soon as we detect the first expected bytecode, we start storing the dumped memory in *searchBuf[k]*. Then we check if the next bytecode is the one expected. If not we reset the index k, and we continue until we have parsed the entire memory. In fact, the Trojan relies on the know-how of the internal representation of the objects in the memory. Due to the fact that this information is not public, we had to get the complete internal structure of the CAP file inside the card. Thus, the search and replace method takes this information into account to locate the expected applet referenced by its AID. It also contains the methods used to get the address of the array, as well as a method to setup the initial value of the array.

The greatest difficulty is to modify correctly the CAP file while keeping all the interdependent information correctly. We do not have a tool performing this in an automatic way (which would be very helpful to handle efficiently such attacks).

5 Evaluation of the attack

The Java Cards that have been considered in this paper are publicly available at some web store. We evaluated some cards from six smart card providers and we will refer to the different cards using the reference to the manufacturer

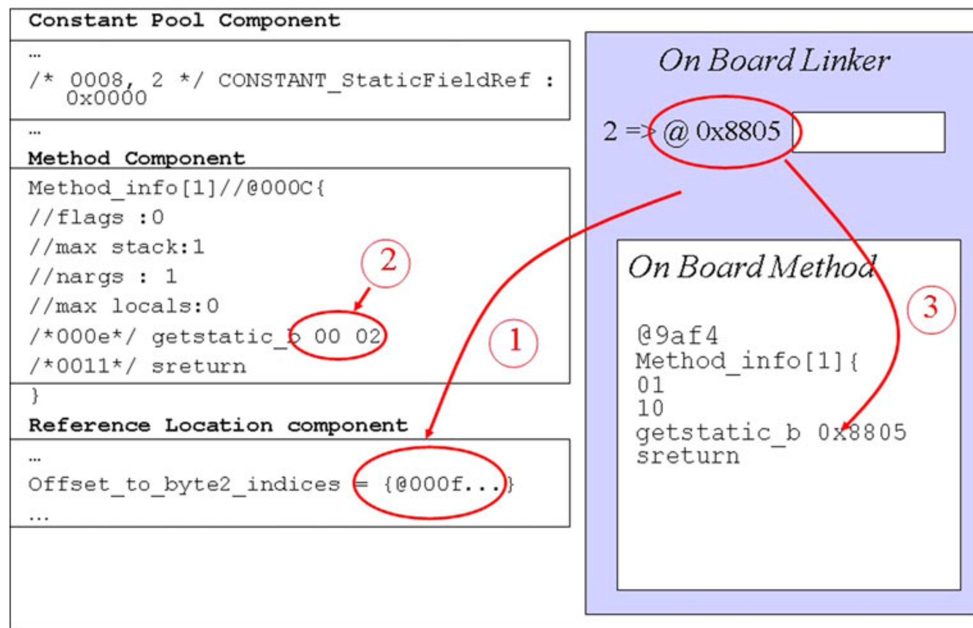

Fig. 1 Embedded linking process

Table 1 Smart Card characteristics used in this study

Reference	Java Card	GP	Characteristics
a-21a	2.1.1	2.01	
a-21b	2.1.1	2.0.1	Same as a-21a plus RSA
a-22a	2.2	2.1	64 KB Eeprom, SLE66CX322 (8051 derivative), page size 1 KB
a-22b	2.1.1	2.0.1	32 KB Eeprom, RSA
b-21a	2.1.1	2.1.2	16 KB Eeprom, RSA P8WE5017 (core 8051)
b-22a	2.1.1	2.0.1	16 KB Eeprom, hW DES
b-22b	2.1.1	2.1.1	P5CD036 (core 8051), page size 2 KB
c-22a	2.1.1	2.0.1	RSA P8WE5033 (core 8051)
c-22b	2.2	2.1.1	64 KB Eeprom, dual interface, RSA
d-22	2.2	2.0.1	
e-21	2.1.1	2.0.1	16 KB Eeprom, SLE66CX165P (8051 derivative), page size 1 KB

associated to the version of the Java Card specifications. At the time we perform this study no Java Card 3.0 were available and the most recent cards we had were Java Card 2.2.

- Manufacturer A, cards a-21a, a-21b, a-22a and a-22b. The a-22a is a USIM card for the 3G, the a-21b is an extension of a-21a supporting RSA algorithm, and the a-22b is a dual interface card.
- Manufacturer B, cards b-21a, b-22a, b22b. The b-21 supports RSA algorithm, the b-22b is a dual interface smart card.
- Manufacturer C, cards c-22a, c22b. The first one is a dual interface card, and the second support RSA algorithm.
- Manufacturer D, card d-22.
- Manufacturer E, cards e-22.

The cards have been renamed with respect to the standard they support. The following table summarizes it (Table 1).

We have conducted this attack on some publicly available evaluation smart cards. While some of these cards implemented countermeasures against this attack, we managed to easily circumvent a few of them.

5.1 Loading time countermeasures

The loader-linker can detect basic modifications of the cap file. Some cards can block themselves when erasing an entry in the `refLocation` component without calculating the offset of the next entry. For instance, the card `a-21a` blocked when detecting a null offset in the `refLocation` component. But it is easy to bypass this simple countermeasure

Table 2 Load time countermeasures

Card reference	RefLocation correct	Type verification
a-21b	x	
c-22b, e-21		x

with elaborate tool able to perform more complex Cap file modifications.

At least three of the evaluated cards have a sort of type verification algorithm (a complex type inference). They can detect ill-formed bytecodes, returning a reference instead of a short for instance. Looking at Common Criteria evaluation reports, it is evident that these cards were out of our hypotheses: they include a bytecode verifier or, at least, a reduced version of it. Thus, such cards can be considered as the most secure, because once the CAP file is detected as ill-formed, cards can reject the CAP file or become mute (for instance `c-22b`) (Table 2).

5.2 Runtime countermeasures

For the remaining cards which pass the loading phase, we can evaluate the different countermeasures done by the interpreter.

A countermeasure consists in checking writing operations. For instance, when writing to an unauthorized memory area (outside the area dedicated to class storage) the card can be blocked or return an error status word. More generally, the cards can detect illegal memory access depending of the accessed object or the bytecode operation. For instance, one card (`c-22a`) limits the possibility to read arbitrary memory location to seven consecutive memory addresses (Table 3).

On the remaining cards, we were able to access and completely dump the memory. The following table summarises the different results we obtained. For each evaluated card, we explain what we have reached with the attack, and then the level of the countermeasure and the portion of the memory dumped (Table 4).

We can compare the countermeasures encountered in this attack with those described in [7]. The first countermeasure described consists in dynamic type inference, i.e. a defensive virtual machine. We never found such a countermeasure on the card we evaluated but may be it is integrated on cards like `c-22b` or `e-21` for which we did not succeed with our attack. Due to the fact that our attack does not modify the array size, any countermeasure trying to detect a logical or a physical size modification will not be efficient. The last countermeasure described concerns the firewall checks. The authors do not try to bypass the firewall using this methodology, thus they did not succeed in discovering this weakness. Nevertheless, their approach could be used, and in particular the buffer overflow for the card `c-22a` for which our attack did not succeed. But if we modify the size of the array, we will be able to bypass the countermeasure on bound check.

Table 3 Runtime countermeasures

Card reference	Memory area check	Memory management	Read access
a-22a		x	x
b-22b	x		
c-22a			x

5.3 Evaluation of other attacks

One of our hypotheses is that the card does not embed a type verifier. In order to relax this hypothesis we evaluate the approach described in [7]. Poll et al. presented a quick overview of classics attacks available on smart card and gave some countermeasures. We will firstly present the different kinds of attacks and after explain which ways we followed.

There are different methods presented in this paper:

- Shareable interfaces mechanisms abuse
- Transaction Mechanisms abuse

The idea to abuse shareable interfaces is really interesting and can lead tricking the virtual machine. The main goal is to have type confusion without the need to edit CAP files. To do that, we have to create two applets which will communicate using the shareable interface mechanism. To create type confusion, each of the applets will use a different type of array to exchange data. During compilation or on load, there is no way for the BCV to detect such a problem.

The problem seems to be that every card they tried, with an on card BCV, refused to allow applets using shareable interface. As it is impossible for an on card BCV to detect this kind of anomaly, Erik Poll emitted the hypothesis that they decided to forbid any use of shareable interface on card with an on card BCV. In our experiments, we succeed to pass a byte array as a short array in all case but when we exceeded the standard ending of the array, an error was checked by the card. This means that the type confusing is possible but a runtime countermeasure is implemented again this attack (Table 5).

The second option was the transaction mechanism. The purpose of transaction is to make a group of operation becomes atomic. Of course, it is a widely used concept, like in databases, but still hard to implement. By definition, the roll-back mechanism should also de allocate any objects allocated during an aborted transaction, and reset references to such objects to null. However, they find some strange cases where the cards keep the reference of objects allocated during transaction even after a roll back. If we can get the

Table 4 Comparison of the countermeasures for the memory dump

Card reference	Reading an address	Writing an address	Countermeasures	Memory dumped
a-21a	x	x		8000-FFFF
a-21b	x		Card Terminated	8000-FFFF
a-22a	x		Bypassed	8000-FFFF
a-22b	x	x		8000-FFFF
b-21a	x	x		8000-BFFF
b-22a	x	x		8000-BFFF
b-22b	x	x		8000-FFFF
c-22a	x		Partially bypassed	Seven bytes
c-22b			Strong	
d-22	x	x		8000-BFFF
e-21			Strong	

Table 5 Array bounds check

Card reference	Type confusion	Result after exceeding array's length
a-22a	Yes	6F 00
b-21a	Yes	6F 00
b-22b	Yes	6F 00
c-22a	Yes	6F 08

Table 6 Abusing transaction mechanism

Card reference	Call to new	Call to Make TransientArray	Type confusion
a-22a	No	Yes	Yes
b-21a	Yes	Yes	Yes
c-22a	No	Yes	No
e-21	No	Yes	No

same behaviour, it should be easy to get and exploit type confusion.

The other confusion we used is an array of bytes and an object. If we put a byte as first object attribute, it is bind to the array length. It is then really easy to change the length of the array using the reference to the object (Table 6).

As observed by [7] several cards refuse a code that creates a new object in a transaction. But surprisingly if we use the method of `MakeTransientArray` of the API it becomes feasible for the cards under test.

6 Future works

During the dump of some cards we discovered that we have had access to a RAM area at the lowest addresses in memory. Firstly, we discovered the reference of the APDU's class instance by using the same method as EMAN: 0x01D2 (situated in RAM area). At this address, the following structure was found: 00 04 29 FF 6E 0E. It represents the instance of APDU class, so we can deduce the address of the class APDU which is 0x6E0E (situated in ROM area).

After this observation, we wanted to find the APDU buffer in the RAM memory which is probably near to the class APDU's instance reference. That's why we have searched a table of 261 bytes (105 in hexadecimal). We found it at the address 0x1DC. It was confirmed because a pattern of an APDU command was found at the beginning of the table: "80 31 00 00 02".

Secondly, we wanted to find the stack. We believed it was near to the APDU buffer. So, we analyzed the operations used when the dump was made and looked in RAM memory. After that, we deducted that the stack was just before the APDU buffer, near to the address 0x7B. In fact, near to this we found this short value 0x01D2 which matches to the instance's reference of the APDU class and 0x01DC which is the address of the APDU buffer.

So we know that the Java stack is implemented on the a-21a card (no experiment has been conducted yet on the other cards). This means that we have the call history available and that the VM doesn't erase the value on top of the stack. We have to check if this is still valid during a context switch. If this hypothesis is true it becomes possible to steal the value of the PIN code of the user. It is passed as a parameter on top of the stack before the call to the method verifyPin().

Finally we want to evaluate the implementation of the BCV, because this component is known to be highly complex and prone to implementation errors. We are developing a model of this security function and a library to manipulate CAP file. Then thanks to our OPAL library,[1] we will generate test suites to characterize the BCV and check faulty implementation.

[1] http://gforge.inria.fr/projects/opal/

More generally, we believe that using various logical attacks could lead to information disclosure even for very recent smart cards. Furthermore, our next step is to use hardware attacks combined with logical attacks.

7 Conclusion

We are currently investigating other countermeasures as well as a way to remove one of our hypotheses. For instance, we are looking at the attack described in [7] to circumvent the BCV protection. Another way is to evaluate the implementation of the BCV which is a component known to be highly complex and prone to implementation errors. We believe that using a framework of different logical attacks should reveal BCV implementations flows.

We have shown in this paper the preliminary results of the EMAN attack. Our contribution is to propose a complete and optimized method of Hyppönen idea by allowing a self mutable code in a card, which has never been done until now. As a result, we are able to search and replace any code fragment in the memory, even if this memory segment is protected by the Java Card security mechanism. We have broken the segregation property offered by the firewall. This attack is based on two hypotheses: (i) post issuance is allowed and we have the necessary credentials, (ii) there is no BCV in the card. Our EMAN attack has been successfully tested against several smart cards. We demonstrated that this exploit was successful on some cards, only one became mute while detecting the ill-formed CAP file using a bytecode verifier. This attack has been made possible by the pre link process and the absence of an embedded bytecode verifier. In order to relax the hypothesis on the presence of such a bytecode verifier, we are investigating for another solution based on a hardware fault injection to introduce type confusion in the code stored in the EEPROM memory.

References

1. Virtual machine specification, java card platform, version 3.0, classic edition (2008). http://java.sun.com/javacard/3.0/
2. Global Platform Specification 2.2. http://www.globalplatform.org/specifications.asp
3. Girard, P., Lanet, J.L.: New security issues raised by open cards. Inf. Secur. Tech. Rep. **4**(1), 4–5 (1999)
4. Anderson, R., Kuhn, M.: Tamper resistance: a cautionary note. In: WOEC'96: Proceedings of the 2nd conference on Proceedings of the Second USENIXWorkshop on Electronic Commerce, p. 1. USENIX Association, Berkeley (1996)
5. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The sorcerer's apprentice guide to fault attacks. Proc. IEEE **94**(2), 370–382 (2006)
6. Joint interpretation library application of attack potential to smartcards, v2.1, available at http://www.ssi.gouv.fr/site_documents/JIL/JIL-The_application_of_attack_potential_to_smartcards_V2-1.pdf (2006)
7. Mostowski,W., Poll, E.: Malicious code on java card smartcards: Attacks and countermeasures. In: Proceedings of the Smart Card Research and advanced application conference (CARDIS 2008), pp. 1–16 (2008)
8. Vertanen, O.: Java Type Confusion and Fault Attacks, Lecture Notes in Computer Science, vol. 4326/2006. pp. 237–251. Springer, Berlin (2006)
9. Witteman, M.: Smartcard security. Inf. Secur. Bull. **8**, 291–298 (2003)
10. Hyppönen, K.: Use of cryptographic codes for bytecode verification in smart card environment. Master's thesis, University of Kuopio (2003). Available at http://dx.doi.org/10.1007/978-3-540-69485-4_15