A Dynamic Syntax Interpretation for Java Based Smart Card to Mitigate Logical Attacks

Tiana Razafindralambo, Guillaume Bouffard,
Bhagyalekshmy N Thampi, and Jean-Louis Lanet

Smart Secure Devices (SSD) Team
XLIM/Université de Limoges – 123 Avenue Albert Thomas, 87060 Limoges, France
aina.razafindralambo@etu.unilim.fr,
{guillaume.bouffard,bhagyalekshmy.narayanan-thampi,
jean-louis.lanet}@xlim.fr

Abstract. Off late security problems related to smart cards have seen a significant rise and the risks of the attack are of deep concern for the industries. In this context, smart card industries try to overcome the anomaly by implementing various countermeasures. In this paper we discuss and present a powerful attack based on the vulnerability of the linker which could change the correct byte code into malicious one. During the attack, the linker interprets the instructions as tokens and are able to resolve them. Later we propose a countermeasure which scrambles the instructions of the method byte code with the Java Card Program Counter (`jpc`). Without the knowledge of `jpc` used to decrypt the byte code, an attacker cannot execute any malicious byte code. By this way we propose security interoperability for different Java Card platforms.

Keywords: Smart card, Java Card, Logical Attack, Countermeasure.

1 Introduction

A smart card is a secure, efficient and cost effective embedded system device comprising of a microcontroller, memory modules (RAM, ROM, EEPROM) serial input/output interfaces and data bus. On chip operating system is contained in ROM and the applications are stored in the EEPROM. A smart card can also be viewed as an intelligent data carrier which can store data in a secured manner and ensure data security during transactions. Security issues are one major area of hindrance in smart card development and the level of threat imposed by malicious attacks on the integrated software is of high concern. To overcome this, industries and academia are trying to develop countermeasures which will protect the smart card from such attacks and render secure transactions [4]. Size constraints restrict the amount of on chip memory and a majority of smart cards on the market have at most 5 KB of RAM, 256 KB of ROM, and 256 KB of EEPROM which has a deep impact on software design. The first tier safety relates to the underlying hardware. To resist an internal bus probing, all components (memory, CPU, crypto-processor, *etc.*) are on the same chip which

S.M. Thampi et al. (Eds.): SNDS 2012, CCIS 335, pp. 185–194, 2012.

is embedded with sensors covered by a resin. Such sensors (light sensors, heat sensors, voltage sensors, *etc.*) are used to disable the card when it is physically attacked. The software is the second security barrier. The embedded programs are usually designed neither for returning nor for modifying sensitive information without guaranty that the operation is authorized.

All applications stored in the smart card should be resistant to attacks. It is important to analyze all the possible attack paths and find a way to mitigate them through adequate software countermeasures. In this paper we are talking about logical attacks where we are abusing the linker to change the correct byte code instruction of a given method into a malicious one. It occurs when a smart card is operating under normal physical conditions, but sensitive information is gained by examining the bytes going to and from the smart card [13].

Developing Java Card application remains a challenge for security purpose. Smart card manufacturers are differentiated from one another by the way they implement security features. An application proved secure on a platform can be prone to hardware attacks on another platform. This difference in security implementation raises serious problems for the certification process like Common Criteria [11]. One of the challenges is to define a common behavior in term of security. For that purpose it has been proposed to define an API or annotation process [4] that could standardize the security behavior of the platform. Within this approach it becomes possible to have a common security behavior of Java Card applications.

This paper is organized as follows: the first section is about Java Card security. The second section provides a brief state of the art on Java Card attacks. In the third section we introduce the new logical attack. The countermeasure which we proposed is suitable for security interoperability and is described in the fourth section. Finally we conclude our work with the future perspectives.

2 Literature Survey

2.1 Java Card security

Java Card is a kind of smart card that implements the standard Java Card 3.0 [12] 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 [7]. Java Cards have shown an improved robustness compared to native applications with respect to many attacks. They are designed to resist numerous attacks using both physical and logical techniques. To resist such attacks several mechanisms have been added while others have been removed from the Java Card specification.

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) run time 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 byte code verifier (invoked by a converter) is executed off-card; while the interpreter, the API and the Java Card Run time Environment (JCRE) are executed *on-card*. 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 into the card. The byte code converter converts the Java class files and verified by a byte code verifier into a CAP file format which is more suitable for smart cards. 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). In order to avoid it, the Global Platform Security Domain checks the integrity and authenticates the package before its registration in the card. Through out this paper, discussion on Java Card refers to the *Classic Edition*.

Element of the Security. The Java Card platform is a multi-application environment in which an applet's critical data must be protected against malicious access from the other applets. To enforce protection between applets, traditional Java technology uses type verification, class loaders and security managers to create private name spaces 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.

CAP File. The CAP (`Converted APplet`) file format is based on the notion of interdependent components that contain specific information from the Java Card package. For example, 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 order to manipulate the instructions of a given method we need to use the `Method` component, which provides all the methods used in the applet and each one contains set of instructions. One optional component (`custom` component) can be used to define proprietary properties on the application like annotation.

Byte code verification. 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 (hand made byte code) 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 byte code level. 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. Byte Code 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, byte code verification is a complex process involving elaborate program analysis. 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 organization that signs the code of the applet to ensure that the code is well typed.

The Linking step. The linking step is defined by the Java Card Specification [12] and has been done during the loading of a CAP file. When the software is loaded into the card, the JCVM provides the way to link the CAP file with the installed Java Card API, thanks to the token link resolution referred in the `Constant Pool` component. Indeed the `Reference Location` component keeps a list of offsets in order to easily retrieve each token placed in the `Method` component.

The 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). If two applets are instances of classes coming from the same Java Card package, they are considered to be in the same context. Every object is assigned to a unique owner context which is the context of the applet that created the object. It is this context that decides whether the 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 attributes or methods of objects belonging to another context.

2.2 Attacks against Java Card platform

There are three main types of attacks on a smart card. First one is the the logical attack, which provides a cheap solution to access sensitive information from the targeted cards. Next is the side channel attack, by which the attacker can obtain the cryptographic secrets [6] with some electromagnetic curves or can find the executed byte code as explained in [1]. The third is the physical attack, which can provide information about the target, optical or laser faults. This sort of physical modification may create a logical fault which is used to attack a card and is called combined attack [3,5]. In this paper, our focus is limited to logical attacks.

First Logical Attacks. E. Hubbers *et al.* presented in [9] a quick overview of the classical attacks available and suggested some countermeasures.

First, a manipulated program is sent to the card. Then it is modified to bypass the BCV after the compilation step. The efficient way to block this attack is an *on-card* BCV. Another solution to have a type confusion without the modification of the applet files is the Shareable interfaces mechanism. The authors created two applets which exchange information, thanks to the Shareable interface mechanism. To create a type confusion, each applet uses a different type of array to exchange data. During compilation or on the loading step, the BCV cannot detect an incoherence. This attack is no more applicable on the new cards. Finally, the last attack is about the transaction mechanism. The aim of a Java Card transaction is to make atomic operations with a rollback mechanism which should deallocate any allocated objects during the aborted transaction and clear the references. On some card, the authors found a way to keep the reference to objects allocated during transaction even after a rollback.

The First Trojan in a Smart Card. In [10], J. Iguchi-Cartigny *et al.* described the way to install a Trojan in a smart card. This Trojan will read and modify the smart card memory. The firewall mechanism is abused to build this attack, thanks to the unchecked instructions on static. These instructions without checks during the installation step, define a way to call a malicious byte code which is presented in a Java Card array. They achieved this through the following steps. The first step obtains the array address and this reference address of the applet instance to be modified. In the second step, the `getstatic` and `putstatic` instructions are used to read and write the smart card memory. Finally, a modification of the `invokestatic` parameter provides the redirection of the install program's Control Flow Graph (CFG). When the `invokestatic` instruction is called, the Java Card Program Counter (`jpc`) jumps to the malicious byte code contained in the Java Card array.

A Java Card Stack Overflow. G. Bouffard *et al.* described in [5], two methods to change the Java Card CFG. The first one, EMAN 2 provides the way to change the return address of the current function. This information is stored in the Java Card stack header. When the malicious function exits on correct execution, the program counter returns to the instruction which addresses it. The address of the `jpc` is stored on the Java Card Stack header. An overflow attack has succeeded to change the return address by the address of our malicious byte code.

Our malicious method has one local variable which received the return of `getMyAddress` function. The function return increased by the size of the Java Card array header (here 6), corresponding to the address of the shell code.

After the characterization of the Java Card stack, the return address was located. In order to modify this address the parameter of the `sstore` instruction was changed. As there is no runtime checking on the parameter it allows a standard buffer overflow attack.

How to find the Java Card API. The main difficulty to use this attack in the previous case is that, there is no access to the linked Java Card API. Hamadouche *et al.* described in [8] a way to abuse the Java Card linker in order to obtain the Java Card API. Some instructions are followed by tokens. These tokens are referred in the constant pool component of the unlinked applet. Speed of the linking process can be increased by using the reference location. The aim of their attack was to resolve the tokens which were preceded by instruction that pushes a short[1] on the stack. So it is easy to send this value into the APDU buffer or update a shell code contained in a Java Card array.

3 Building a New Attack

Assume that there is no embedded *on-card* BCV. In order to characterize this card, it is necessary to abuse the previously explained linking mechanism.

To perform this attack, there are set of instructions which are to be modified by using an abuse linking mechanism as shown in the listing 1.1. We used a

[1] On the targeted card, each address are stored into 2-byte value

tool developed by our team: the `Cap Map` [14], for CAP File Manipulator. It provides a friendly environment to modify the CAP file by respecting the inter-dependencies between the affected components.

Each instruction is referred by an offset in the current method in the `Method` component.

```
/*0020*/    [0x00]          nop
/*0021*/    [0x02]          sconst_m1
/*0022*/    [0x02]          sconst_m1
/*0023*/    [0x3C]          pop2
/*0024*/    [0x04]          sconst_1
/*0025*/    [0x3B]          pop
```

Listing 1.1. Set of instruction to attack with the link mechanism abuse technic

As we have previously seen in the section 2.1, it is the `Reference Location` component that helps to link between a token used in the `Method` component and the `Constant Pool`.

By using the linking mechanism abuse, described in the section 2.2, the linker uses the instructions `nop` and `sconst_m1` (`0x0002`) as a token.

```
.ConstantPoolComponent {
    [...]
    /* 0008, 2 */CONSTANT_StaticMethodRef:
                        external: 0x80, 0x8, 0xD
    [...]
}

.ReferenceLocationComponent {
[...]
    offsets_to_byte2_indices = {
        [...] @0020 [...]
    }
}
```

Listing 1.2. Reference Location modification with CAPMAP

To perform the previous manipulation as seen in the listing 1.2, we modify the `Reference Location` by adding a new link. The offset value `0x0020` is referred to the token `0x0002` in the `Method` component. In the `Constant Pool` component, we see that this token is associated to a static method reference. By looking to our first linker attack presented in the section 2.2, the token method `0x0002` is linked by the value `0x8E03` into the targeted card.

Once the Java Card linker finished linking as shown in the listing 1.3, it mutates the method byte code. The link resolution needed two bytes, and the instructions from the offset `0x0021` to `0x0024` became the `invokeinterface` operands.

```
/*0020*/   [0x8E]          invokeinterface
/*0021*/   [0x03]              // nargs
/*0022*/   [0x02]              // indexByte1
/*0023*/   [0x3C]              // indexByte2
/*0024*/   [0x04]              // method
/*0025*/   [0x3B]          pop
```

Listing 1.3. Set of instruction after link resolution

In this case abusing the token resolution mechanism leads to the call of a method referred in the `Constant Pool` by the index composed of two bytes `0x02`, `0x3C`. This index corresponds to the method `getKey` which gives us the ability to return the key data *via* the APDU buffer. Most of the attack in the literature tried to retrieve the secret key thanks to physical means. Here it is possible to force the virtual machine to send back clear text value of the key to the attacker. Of course, this attack works well due to the absence of the byte code verifier which could have detect the ill formed CAP file. But as demonstrated by G. Barbu in [2] a laser fault can allow logical attack with or without the presence of the byte code verifier. Therefore the shell code do not access any objects of the security context and it will never detect the attack.

4 The Newly Proposed Countermeasure

G. Barbu in [2] proposed a countermeasure which prevents the malicious byte code from being executed. His idea was to scramble each instruction during the installation step. For that each Java Card instruction *ins* performs a xor with the $K_{bytecode}$ key. Thus the hidden instructions are computed as follows:

$$ins_{hidden} = ins \oplus K_{bytecode} \tag{1}$$

If an attack as EMAN 2 succeeds described in the section 2.2, the attacker cannot execute his malicious byte code without the knowledge of the $K_{bytecode}$ key. Thus to find the xor key he should just change the CFG of the program to a `return` instruction. As defined by the Java Card specification [12], the associated opcode is `0x7A`. With a 1-byte xor key, this instruction may have 256 possibles values. A brute force attack offers the way to find the xor key.

To improve his countermeasure we add a `jpc` value to perform the hidden instruction and it can be written as:

$$ins_{hidden} = ins \oplus K_{bytecode} \oplus jpc \tag{2}$$

By using the previous example described in the section 2.1, we scramble the byte code in the installed method to prevent a modification of the original byte code from an attacker. For that, we have a $K_{bytecode}$ set to $0x42$. So in the installed applet, we have the following byte code as given in the listings 1.4 and 1.5 given below.

```
/*0x8068*/ 0x42 nop
/*0x8069*/ 0x40 sconst_m1
/*0x806A*/ 0x40 sconst_m1
/*0x806B*/ 0x7E pop2
/*0x806C*/ 0x46 sconst_1
/*0x806D*/ 0x79 pop
```

Listing 1.4. Scrambling Byte Code with the equation 1

```
/*0x8068*/ 0x2a nop
/*0x8069*/ 0x29 sconst_m1
/*0x806A*/ 0x2a sconst_m1
/*0x806B*/ 0x15 pop2
/*0x806C*/ 0x2d sconst_1
/*0x806D*/ 0x12 pop
```

Listing 1.5. Scrambling Byte Code with the equation 2

In the listing 1.4, the scrambling was done without the `jpc` value. If you have many times the same instruction `sconst_m1` in the example will always have the same value. Thus it becomes easy for an attacker to find this constant key value (to find it, an attacker has a constant complexity in $O(256)$). To improve that, we added the `jpc` value. As described in the listing 1.5, each similar instruction has a different byte code value.

This countermeasure can be enabled by the developer during the compilation step. For that he has to set each enabled countermeasure flag on CAP file custom component. It is only parsed if the targeted JCVM can parse it. The way to enable or not this countermeasure on specific applet provides an additional complexity for the attacker. An special key for each security context may be used to improve this protection. For the Java Card runtime this countermeasure is not expensive. Indeed, just a double xor should be done at the beginning of the main loop. A native implementation is provided in the listing 1.6.

```
while (true) {
   if (scrambled) ins = ins_array[jpc] ^ Key ^ (jpc & 0x00FF
       )
   else           ins = ins_array[jpc]
   switch (ins) {
      case ...
      /* a case for each Java Card instruction to execute it
         in which is incremented */
   }
}
```

Listing 1.6. A countermeasure implementation

Depends on the `jpc` value, each instruction is stored in the smart card memory. Without the knowledge of where each instruction is stored in the EEPROM, an attacker will not have the possibility to execute some malicious byte code by the attacked JCVM.

If an attacker succeeded to change the return address, he can jump to the shell code to read Java Card memory as described in the section 2.2. For that, he use the unscrambled shell code as given in the listing 1.7. This shell code is stored in the EEPROM at the address 0xAB80. The information in the array should not be masked by the Java Card loader like the executed instructions.

```
/*0xAB80*/ 0x8D getstatic 8000
/*0xAB83*/ 0x78 sreturn
```

Listing 1.7. Unscrambling shell code

```
/*0xAB80*/ 0x4F sshl
/*0xAB81*/ 0x43 ssub
/*0xAB82*/ 0xC0 //
    Undefined
/*0xAB83*/ 0xB9 //
    Undefined
```

Listing 1.8. Unscrambling shell code

During the execution of our shell code, the runtime unmasks each instruction to obtain the code shown in the listing 1.8. Of course the code is detected invalid by the interpreter because `0xC0` is undefined by virtual machine. Moreover the `sshl` and `ssub` byte codes need two parameters on the top of the stack. Thus the interpreter will detect an empty stack.

5 Conclusion and Future Work

This paper contributes a way to protect the Java Card from logical attacks. We introduced a powerful logical attack based on the linker vulnerability. This attack allows one to execute a buffer overflow attack on a smart card and it succeeds well with several products which demonstrates the need of an efficient countermeasure. We proposed a cost effective countermeasure to mitigate this attack. This countermeasure scrambles the binary code. Within this process the syntax of the stored code varies according to a variable. Reverse engineering the executable code becomes impossible if the scrambled memory is dumped. Attacker cannot execute the malicious byte code without knowing where the application instructions are stored in EEPROM. Our future work involves reverse engineering process using the electromagnetic side channel attack [15] and evaluate the ability to bypass the proposed countermeasure.

Acknolwledgements. The work was partly funded by the French project INOSSEM (PIA-FSN2 - Technologie de la Sécurité et Résilience des Réseaux) and the Région Limousin.

References

1. Aranda, F.X., Lanet, J.L.: Smart Card Reverse-Engineering Code Execution Using Side-Channel Analysis. NTCCS, Théorie des Nombres, Codes, Cryptographie et Systèmes de Communication, Oujda, Marocco (April 2012)
2. Barbu, G.: On the security of Java Card platforms against hardware attacks. Ph.D. thesis, TÉLÉCOM ParisTech (2012)
3. 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)

4. Bouffard, G., Lanet, J.-L., Machemie, J.-B., Poichotte, J.-Y., Wary, J.-P.: Evaluation of the Ability to Transform SIM Applications into Hostile Applications. In: Prouff, E. (ed.) CARDIS 2011. LNCS, vol. 7079, pp. 1–17. Springer, Heidelberg (2011)
5. Bouffard, G., Iguchi-Cartigny, J., Lanet, J.-L.: Combined Software and Hardware Attacks on the Java Card Control Flow. In: Prouff, E. (ed.) CARDIS 2011. LNCS, vol. 7079, pp. 283–296. Springer, Heidelberg (2011)
6. Clavier, C.: De la sécurité physique des crypto-systèmes embarqués. Ph.D. thesis, Université de Versailles Saint-Quentin-en-Yvelines (2007)
7. Global Platform: Card Specification v2.2 (2006)
8. Hamadouche, S.: Étude de la sécurité d'un vérifieur de Byte Code et génération de tests de vulnérabilité. Master's thesis, Université de Boumerdés (2012)
9. 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)
10. Iguchi-Cartigny, J., Lanet, J.: Developing a Trojan applet in a Smart Card. Journal in Computer Virology (2010)
11. ISO/IEC: Common Criteria v3.0. Tech. rep., ISO/IEC 15408:2005 Information technology - Security techniques - Evaluation criteria for IT security (2005)
12. Oracle: Java Card Platform Specification
13. Petri, S.: An introduction to smart cards. Secure Service Provider TM (2002), `http://www.artofconfusion.org/smartcards/docs/intro.pdf`
14. Smart Secure Devices (SSD) Team – XLIM, Université de Limoges: The CAP file manipulator, `http://secinfo.msi.unilim.fr/`
15. Vermoen, D., Witteman, M., Gaydadjiev, G.N.: Reverse Engineering Java Card Applets Using Power Analysis. In: Sauveron, D., Markantonakis, K., Bilas, A., Quisquater, J.-J. (eds.) WISTP 2007. LNCS, vol. 4462, pp. 138–149. Springer, Heidelberg (2007)