J Comput Virol Hack Tech (2017) 13:109–123
DOI 10.1007/s11416-016-0276-0

ORIGINAL PAPER

Chronicle of a Java Card death

Mozhdeh Farhadi[1] · **Jean-Louis Lanet**[2]

Received: 10 December 2015 / Accepted: 24 April 2016 / Published online: 17 May 2016

Abstract Various attacks are designed to gain access to the assets of Java Card Platforms. These attacks use software, hardware or a combination of both. Manufacturers have improved their countermeasures to protect card assets from these attacks. In this paper, we attempt to gain access to assets of a recent Java Card Platform by combining various logical attacks. As we did not have any information about the internal structure of the targeted platform, we had to execute various attacks and analyze the results. Our investigation on the targeted Java Card Platform lead us to introduce two generic methods to gain access to the assets of Java Card Platforms. One of the new methods we present in this paper is based on the misuse of the Java Card API to build a type confusion and get access to the objects (including cryptographic keys) of a Java Card applet. The other method is a new approach to get access to the return address of the methods in Java Cards with *Separate Stack* countermeasure. We also propose a pattern that the targeted platform uses to store data and code of applets on the card plus the ability to read and write in the data and code area of the applets in different security contexts. These new attacks occur even in the presence of countermeasures such as *Separate Stack* for kernel and user data, indirect mapping for objects addressing and firewall mechanisms.

Keywords Java Card Platform · Attacks · Memory layout · Type confusion · Cryptographic key · Change Flow Control Attack · Separate Stack · Frame overflow and stack underflow · Countermeasures

✉ Mozhdeh Farhadi
mojde.farhadi@gmail.com

Jean-Louis Lanet
jean-louis.lanet@inria.fr

[1] Tehran, Iran

[2] INRIA, LHS-PEC, 263 Avenue Général Leclerc, 35042 Rennes, France

1 Introduction

The smart card is a plastic card equipped with a secure microcontroller. These cards are mostly used by applications that require some user specific data to be stored and retrieved in a secure way, e.g. identity, banking or e-passport applications. In most cases, the card may store some sensitive personal data, such that it becomes necessary to protect this data from unauthorized access. Smart cards provide several assets like PIN, keys, cryptographic algorithms and processes to protect the stored sensitive data from outer world. Accessing the data is only possible when the accessing entity proves that it is authorized to have access to them. Unfortunately most cards are prone to attempt to retrieve these assets. Many efforts have been made by the smart card manufacturers to increase the security of cards to mitigate these attacks. In this situation, there is a type of smart cards called Java Cards which suffer more attacks due to the possibility to load programs after issuance. Evaluating the risk and thus the security of such a product is of a prime importance.

We have developed in the past several attacks [7,16,19] that target different elements of Java Cards. In this paper, our goal is to evaluate how a given Java Card mitigates these attacks and to identify possible ways to break the security and get information about the Java Card Platform. We use the paradigm of attack tree [7] as a generic representation of our know-how on the different scenarios to attack the card. The attack tree also represents the mitigation mechanisms implemented. All these possibilities are possible attacks, we need to characterize the platform to check if a given scenario can be instantiated upon it. Then, we use experiments

to define which branch of the tree can be used to insert hostile code into the card in order to dump the memory.

This methodology helps us to optimize the attack (minimizing the time spent to search vulnerabilities). At a first glance, the card mitigated most of our attacks except one: the basic `getStatic`. This attack is no longer used, requiring too much time to dump the memory: 24 h compared to the couples of minutes of an up-to-date attack for dumping memory of the card. Nevertheless, we understood several mechanisms used in the card. Analyzing the results, we discovered a new type confusion in the system. The idea is related to a weak implementation of the API `Util` of the Java Card specification [24]. The platform was vulnerable against our new attack which we fully describe in experiments section. This vulnerability has allowed us to dump the memory as expected, but the most important was to perform in-line code injection in the memory. Thus, we bypassed the firewall mechanism, which in turn allows us to read data of other applets and in particular their secret keys. We observed that this card didn't implement secure containers and keys are stored unencrypted in the memory. We also investigated the possibility to generate a buffer overflow and we develop the concept of frame overflow that allows us to overcome the split stack countermeasure.

The chip and the cryptographic libraries of the card we used for our investigations are certified by a major European certification center but the platform itself has no certification. The Java Card Platform has been launched in 2013, and implements some up-to-date countermeasures like separate evaluation stack. We applied our methodology and we observed that this platform is a weak implementation of the specification which includes a weak array bound check and the possibility to use a reference of an instance instead of a reference of an array. This new attack path and the frame overflow has been added to our attack tree. We propose several countermeasures and in particular the concept of secure linker to mitigate the possibility to use the basic `getStatic` attack.

The rest of the paper is organized as follows, first we recall the different elements that participate in the security of the Java Card Platform. Second we present the different software attacks that have been proposed in the literature. Third we introduce our methodology using attack trees. Fourth we describe our contribution and the experiments on the platform to instantiate our methodology. Sixth, we propose countermeasures in order to mitigate our attack and last we conclude in the last section.

2 Java Card security

Java Card is a smart card which embeds a lightened version of the Java Platform, and whose applications follow the programming model of applets. They are developed according to Java Card specifications and thus the applications are independent of specific hardware. A Java Card allows different applications to be resident in separate environments (i.e. security contexts) at the same time. It also allows loading new applets even after card issuance. To install an applet, it should be converted to CAP (Converted APplet) file format and then loaded using a dedicated secure loader.

As a Java Card Platform encapsulates the underlying complexity and details of the smart card system [10], a Java Card applet is normally less complicated than a native program.

The security features of the Java Card Platform is comprised of the security of Java language and the security features defined in the Java Card Platform [10].

The following is a list of Java language security features:

- The Java language is a strongly typed language. The Java compiler checks all operations against Type mismatches.
- The Java language does not allow array indices to access memory out of the allocated array boundaries area.
- The Java language does not allow pointer arithmetic. Thus, The Java language prevents known security risks of C and C++ languages.
- Access to all classes, methods, and fields is controlled in regard to the defined level of access. The level of access can be `public`, `protected` or `private` [27].

The Java Card Platform has security mechanisms such as Byte Code Verifier (BCV), Firewall and GlobalPlatform (GP).

- BCV: The Java Card uses a BCV to ensure that applets respect the semantical constraints. The verification of the applets can be done either on card or off-card. The off-card verification of applet is done after the conversion of the applet to a CAP file. If the file passes the off-card verification process, it can be loaded and installed. It is also possible to sign the verified file before loading it into the card. This ensures that the verified file will not be changed before the loading process started. The verification phase can also be done on card before the CAP file installation. But due to limited resources and the resource consuming nature of the BCV, this feature is optional and most of the time verification is done off-card.
- Firewall: The Java Card Platform uses a firewall mechanism to separate the applets and their access to other applets data or resources. The embedded firewall performs dynamic checks to prevent applets from accessing (reading or writing) data of other applets. When an applet is created, the system uses a unique Applet IDentifier (AID) from which it is possible to retrieve the name of the package in which the applet is defined. From this AID a security context token is derived. If two applets are instances of classes from the same Java Card pack-

age, they belong to the same security context. The firewall isolates the contexts such that, a method running within a given context cannot access any attributes or methods of objects belonging to another security context unless it explicitly exposes features via a Shareable Interface Object (SIO). Thus, at runtime, the interpreter verifies that the owner context of an accessed object is equal (or compatible) to the current context. Under some circumstances, the context can be different, i.e. The runtime can access any object belonging to application contexts if it has specific privileges.

– GP: There exists a standard to manage the life cycle of applets on the Java Card. The GP standard controls the process of loading, installing and deleting applications from the card. For example, the GP checks if the loading process obeys the loading protocol defined in the GP standard, or if the loader is authorized to load new applets into the card or not.

There are also programming rules which must be respected by the applications; those rules are statically checked but are not mandatory (e.g. controlled use of the `getKey` method, no use of the method `setValidatedFlag`,...).

Security certification of a Java Card product comprises the security certification of card hardware, platform and applets loaded on the card. But, to account the relation between its elements, it is necessary to evaluate the security of a device also as a whole system. In order to have a good security analysis, it is beneficial to think as an attacker and try to investigate various methods to threaten the security of the card under evaluation. An attacker can use hardware, software or a combination of these two to break the security of a Java Card.

Smart card security is a complex problem with different aspects. Products based on the Java Card Virtual Machine (JCVM) have successfully passed real-world security evaluations like Common Criteria [11] for major industries around the world. During certification, the evaluators apply state of the art attacks in order to assess the level of resistance of the product. A certified Java Card Platform has passed high level security evaluations, for instance, by banking associations and by leading government authorities. Despite the above drawbacks, Java Cards still remain the more secure device for storing secrets.

3 State of the art of software attacks on Java Cards

Software attacks use software or logical methods to get access to the assets of the card. These attacks are more common than hardware attacks because they do not need professional equipment. Examples of software attacks in the Java Card are: CAP file manipulation, shareable interfaces mechanism abuse and transaction mechanism abuse. An example of hardware attacks is fault attacks which use physical devices or physical manipulation to cause an unplanned behavior of the card which may lead to information leakage [17]. Since the seminal work of E. Poll in [23], where the author made a complete survey on different attacks and their countermeasures several new attacks have raised targeting different type of memory or techniques to dump the memory or execute arbitrary code.

In [1], Barbu et al. proposed a combined attack where in a preliminary phase they use a laser to allow them the execution of an ill-typed applet. Their applet is installed on the card after it has been checked by a BCV. It is then considered as a structurally and semantically valid applet. The aim of their attack is to create a type confusion to forge a reference of an object. The authors also explained the principle of instance confusion, similar to the idea of type confusion where the objective is to confuse an instance of object A to an object B by dynamically inducing a fault using a laser beam during the `checkcast` instruction. As they designed the platform, they have been able to perform easily their attack having a complete knowledge of the virtual machine internals.

Software based attacks require less instrumentation and thus cost quite nothing. It relies on the fact that the code has not been verified or it exploits a vulnerability hidden in it later. A flaw in the BCV has been found by [15] or more recently in [20] that has led to a new version of the BCV.

There are many assets in a card that can be either data or code. Often, one tries to recover or modify the value of keys which is known to be a hard task. But the code can also be an asset to protect. If manufacturers succeed to protect data, it is often much easier to break the integrity of the code as a first step before attacking the data. In the next section, we present some successful approaches to gain access to the code stored inside the card.

3.1 Methods to get access to the EEPROM

There are at least two ways to read the content of the EEPROM memory, using `getstatic` instruction and array extension approach. The `getstatic` instruction allows us to read a memory address with an offset given as a parameter. The offset is in fact a token that the embedded linker resolves during the load process. In order to avoid the linking effect, one can modify the `Reference Location Component` of the CAP file as explained in [19].

Within this approach, by loading such an applet, it is possible to gain access to the content of one memory address. For accessing another one, the attacker must upload a new applet. To overcome this limitation, several approaches have been used to allow the execution of a shell code with self-modifying code, which allows a quick dump of the whole memory. The most used is the EMAN2 attack [7], where the authors use an illegal local variable index to have access to the

return address of the current method, thus allowing them to execute data contained in an array, after the execution of the method completes. The hidden code in the array performs EEPROM dump which leads to information retrieval from the platform. Another way to update the return address is the `sinc` instruction. The `sinc` instruction aims to increase a local short variable by a constant value given in its parameter. Normally, during the installation process, the BCV takes care of checking the index of manipulated local variables. A lack of checking would lead to a successful control flow transfer attack.

In [14], Faugeron presented another way to fool the Java Card runtime based on the `dup_x` instruction. This instruction duplicates the top of operands stack words and inserts them below. If the Java Card operands stack does not contain enough elements, the runtime uses the system data as words for the `dup_x` instruction. Thus, an attacker can shift the value of the frame header by a custom words pushed on the stack.

One can find plenty of avatars of these attacks like `swap_x` instruction. This instruction swaps words on the top of stack. The `swap_x` takes as parameter the value `mn` and swaps `m` words with `n` words. If the stack contains less than `m+n` words, the `swap_x` instruction will make a stack underflow. With the appropriate values, the frame header can be overwritten leading to the execution of a shell code which will dump EEPROM memory.

Of course several countermeasures have been designed to avoid an illegal use of the `getstatic` instruction or control flow attacks. We can cite at least:

- Some old smart cards refused to load applications containing `static` instruction. Even if very efficient, such a card does not implement the specification;
- The pool of static is another encountered countermeasure where only a subset of the address range can be eligible to use with the `static` instruction. The pool can also be implemented with a linked list of static variables;
- The most implemented countermeasure is the control flow check. Within this countermeasure each destination of a jump is checked to remain within the span of the current method; this should mitigate the execution of a shell code in an array;
- Secure design of the on board linker. We have found different degrees in the implementation which can be non-functional, functional but insecure and functional and secure. This will be explained in Sect. 6, it requires to verify carefully the token of the `Reference Location` component;

Recent cards are implementing at least one of these countermeasures in such a way that the control flow attack becomes more difficult.

The race between defender and attacker leads to attacks against these countermeasures. The one related of the jump destination has been bypassed easily with an attack presented in [13] where the authors manage to extract the control flow from the shell code in such a way that only sequential parts stored in an array are executed outside the span of the method. They resume the control flow to the caller with enough information to perform a semantically equivalent program. To bypass the *Separate Stack* counter measure, the authors in [7] use the specificity of the `jsr/ret` instructions to execute a shell code within the span of the current method, thus avoiding the previous mitigation mechanism. Moreover they just need to embed and execute dead code.

The second way to get access to the code, is obtained by array extension attack. If an attacker increases the size of an array, he will succeed in reading more bytes than those contained in the original array. In the first publication [18] on this topic, the authors abuse the shareable interface. The main goal was to obtain a type confusion without the need to modify CAP files. They created two applets which communicate using the shareable interface mechanism and each of the applets uses a different type of array to exchange data. During compilation or at load time, there is no way for the BCV to detect such a problem. The array extension attack has been solved by counting the number of bytes instead of the number of elements of the array. The authors suggested another issue with the transaction mechanism used to make a group of operations become atomic. The specification expresses that the rollback mechanism should de-allocate any object allocated during an aborted transaction, and reset references to such objects to null. However, in some cases the cards keep the references of objects allocated during transaction even after a roll back which allows them to reallocate a new array with a different type (this is a type confusion attack) to get access to the data or code located after. This scenario allowed the authors of [8] to successfully perform a type confusion attack. They have been able to modify the size and the area of a transient array, providing them the ability to dump the entire EEPROM area.

3.2 Access to the ROM area

If accessing the EEPROM area is an easy task, even on recent European smart cards, accessing the ROM area or the cryptographic processor area is more difficult. The first attempt [5] refers to the ability to read the content of the ROM on a rather old model of smart card. The author has found an indirection table in the EEPROM area where at least one method of the API was patched. The patch was stored inside the EEPROM area. He demonstrated the ability to add an entry in this table that refers to a code he inserted written in native code. This code just dumps the content of the ROM by bypassing the virtual machine protections. The author tries to reverse the

code, in such a way that some of the methods have been completely reversed. It seems that the code is not completely reversed due to a lack of time or the lack of reverse engineering knowledge. He shows also that some recent cards suffer from the same issue without being able to exploit it.

In a more recent exploit [20], the authors show that it is still possible in a modern card, to execute native code. The idea is to use the inconsistency of a field in the virtual method table. This allows them to use the header of a method to generate the expected offset. Then, while invoking this method, the control flow jumps where the designer expects, leading to execute a payload. Having an access to the source code of the product, the authors are able to derive the control flow to a native payload executed from the content of the APDU buffer via an indirection in the communication buffer of the NFC chip. This allows them to execute any Advanced RISC Machines (ARM) code, which in turns gives them access to the binary code of the virtual machine and all the counter measures.

4 Methodology for security evaluation

The first step is identifying the assets that the attacker wants to have access to, and then verifying the possibility of threats. From an attacker point of view, the threat analysis can be considered as its know-how on a generic target. There are several methods to represent threats but the attack tree model gives us a global view on the vulnerability of the device. Attack trees have been introduced by Schneier in [26], they form a convenient approach to analyze the different ways in which a system can be attacked. It is an analytical technique (top-down) where an undesirable event is defined and the system is then analyzed to find the combinations of basic events that could lead to the undesirable event. The refinements are combined using conjunctive or disjunctive gates. The undesirable event represents either the objective of the attacker or a property of the asset to protect by the defender. The seminal work of Schneider has been extended to Defense Trees [4], Attack Countermeasure Trees [25], Boolean logic Driven Markov Processes [9] and so on.

With this formalism, it is possible to represent all possible paths allowing the attacker to succeed in reaching the root node. Nevertheless, the attack tree represents only a static view for the attacker, i.e a generic framework for attacking the device. Thus, it needs to be instantiated for a given product. In [7], the authors applied the attack tree analysis to have a global view on the vulnerability of the smart card. Having an attack tree for an asset, we need to verify with experiments if a given branch of the tree is possible, i.e. no countermeasure will mitigate this attack. Experiments form the instantiation of the attack tree on a given product. The assets to be protected are the code and the data. The properties to be checked on each of them are confidentiality and integrity. So, we have four attack trees. In this paper, we will only develop the code confidentiality attack tree which is the conjunction (OR gate) of two subtrees: Control Flow Attack (CFA) and Array Extension Attack (AEA). We will develop in this paper only the CFA subtree. Our new attacks which is presented in this paper, add new branches to both the CFA and AEA subtrees.

Such an analysis is closed to the risk analysis community with the cause-effect diagrams. An attack tree is a tree in which the nodes represent attacks. The root node of the tree is the property that an attacker wants to break. Children of a node are refinements of this goal, and leafs therefore represent initial causes. An attack tree is not a model of all possible combinations but a restricted set. It is related to the evaluated property. In this case, code integrity is the most sensible property, because if it is not guaranteed, it enables the attacker to execute any arbitrary code.

Figure 1 shows the both sides of attack and defense concerning the possibility to execute a shellcode. If the BCV can be bypassed (D5) and if there is no strict implementation of the `Reference Location` verification (D4) and if the usage of the static is not controlled (D6), then possible attacks are either the basic `getStatic` attack (A1) or any attack related to the return addresses. To execute the latter, it requires that the card does not implement a *Separate Stack* (D1) for system and data. If there is no frame integrity (D2), then the Faugeron attack is possible (A3). If it does not implement checks on local values (D3), then the EMAN2 attack is possible (A2). As one can see, the higher in the tree the countermeasure is, the better its coverage is.

A strict implementation of the `Reference Location` check seems obvious, but it is only implemented by a little number of Java Cards. The next step is to characterize if some countermeasures are implemented in order to set up the optimal attack requiring the minimum effort for the attacker. In a perspective of an evaluation lab, the attack tree points out the parts of the software that need to be carefully verified. A simple set of ill-typed applets can verify how the countermeasures are implemented.

5 Experiments

In this section, we describe various experiments designed to retrieve information from internal structure of a Java Card 2.2.2 and GP 2.1 platform. The targeted Java Card Platform has a 32 bit processor, with 256 KB of ROM, 40 KB of EEPROM and 10 KB of RAM. As we have no information about the internal structure of this platform, each experiment improves our understanding about the platform. In order to retrieve information from our targeted Java Card Platform, we follow attacks published in recent scientific papers and define them in the attack tree which is introduced in the previous section. Some of the attacks are not completely applicable because of the countermeasures defined in the Java Card Plat-

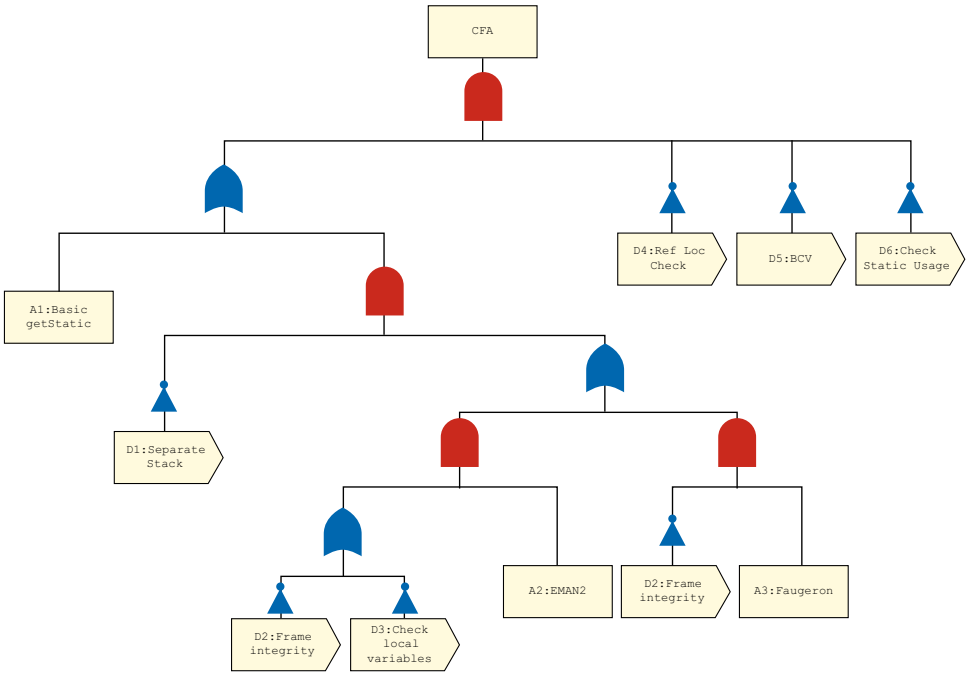

Fig. 1 A CFA with the corresponding countermeasures

form that we are performing experiments on it. Our understanding of the targeted Java Card is mostly based on analyzing results of the card dump which is an exhaustive work, but we show that this method is the main method that applies well in our targeted platform. To perform our experiments on the card, the ability to load and install applets is assumed as a perquisite, thus we used a development Java Card.

5.1 Limitations of the basic method

We consider that the BCV is not used, this hypothesis removes the countermeasure (D5). A simple check verifies that this card does not implement the strict `Reference Location` countermeasure (D4). Then, we try to determine which leaf of the left branch of our tree can be exploited.

As described in Sect. 3.1, one of the methods to dump memory is using `getstatic` op-code (A1). Using `getstatic` op-code with different memory addresses as operands to dump the card has some drawbacks. For example: a high stress on the two memory cells that store operands of `getstatic` op-code and also low speed of dump operation. The authors in [19] proposed a new technique (A2) to read memory content of different addresses using a malicious code which is stored inside a transient array. In their method, `getstatic` op-code and its operands are stored as elements of an array which is called malicious array. In order to read memory content of a desired address, we only need to change the two array elements which correspond to the operands of `getstatic` op-code. To complete this attack, it is needed to redirect program flow to the content of the malicious array using `invokestatic` op-code leading to a self-modifying code. Thus we need to know the reference of this malicious array first. In our targeted platform, we intend to use the described technique to dump the card and then to analyze the results. We get a reference of the malicious array and then change array elements corresponding to `getstatic` op-code in malicious array. After several experiments, we conclude that memory content at returned reference does not contain malicious array data. Also reading memory content of addresses around this reference do not contain our malicious array data. As we do not have access to the actual reference of the malicious array in our applet code, the methods that are based on finding a reference of an array to redirect program into its content will not work. As the mentioned smart approaches described in [7] and [14] do not work with our platform, we decide to dump whole memory using `getstatic`. We search the content of our malicious array in the dump to find relation between array reference and the actual reference that the array elements are stored there. As the platform did not implement the pool of static countermeasure (D6), we are able to use `getstatic` op-code with various addresses to read memory content of them. Thus, we start with a static array and search for its content in the memory.

5.2 Access to static data

Based on the experiments, we could not get access to the content of an array in our targeted platform. Thus, in this section we try to dump the whole memory of the card in order to find the relation between the returned reference of an array and the actual reference where the array elements are stored. We write an applet which uses `getstatic` op-

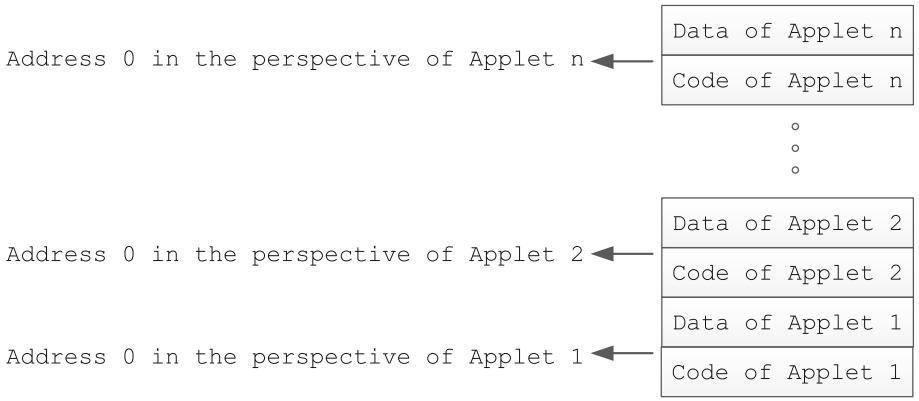

Fig. 2 Memory layout of the targeted platform

code and then modify the corresponding CAP file in order to read the memory content at the desired addresses. The `getstatic` op-code has two operands which construct an index into the constant pool component of a package. As we want to directly read memory contents at addresses that are presented as operands of `getstatic` op-code, we cut the link between `getstatic` operands and items of Constant Pool component (bypass D4). We remove this linkage data which is stored in the Reference Location component. After these modifications of the CAP file, we obtain an applet which has memory dump capability for desired addresses in the range of 0 to 65535. We insert a static array in our applet and fill the array with some distinct values in order to search the dump result for these indicated values. In the dump result of the card, we find the area where the code of our applet is stored, but we can not find data of the defined static array. Thus, two hypotheses are raised. The first one is that the data of the static array are encrypted. The second hypothesis is that array data are stored in plain in the card but not in the scope of our dump. In order to check the first hypothesis, we change the value of our static array and dump the card again to see if the dump result is changed. But in the dump result we do not observe any remarkable change. So the second hypothesis should be tested. In our experiments, we did not find any trace of the array's data in the area that the reference of the static array were pointing to it. Thus, we conclude that there might be an indirection table that maps these references into other addresses where the array data is actually stored.

5.3 Get access to the other security contexts

In this section, our objective is to change the scope of our dump by dumping memory of the card using another applet. With this method, we want to investigate if the second raised hypothesis in the previous section is correct. Thus, we install another applet with the same code in another package while the previous applet also exists on the card. We dump memory of the card from address 0 to 65535 using the second installed applet. At the dump result, we see applet code of the first and second installed applet. As AIDs in the CAP file of the two applets are different, code of applets can be distinguished from each other. In the dump result, we recognize an area which contains the static array data. As the content of the static array for the two installed applets are the same, we iterate the experiment with a change in the array content of the applets to verify to which applet belongs the found array data. It reveals that the found array content belongs to the firstly installed applet in the card.

We repeat this experiment with more than 2 applets in different cases. Finally we conclude that the data of an applet is stored at places which are non-accessible for the applet. So, if we start dumping the card using `getstatic` op-code from address 0, the first thing we get is the code of the applet, while the data of the applet is stored in addresses before the code section. So, this is the reason why the data of the applet is not accessible in the dump result of the dumper applet. This inability to obtain the applet's data can be considered as a countermeasure. The Fig. 2 represents layout of the applets in the targeted Java Card Platform.

We conclude that, while we may not be able to observe the static data related to our applet in the dump, it is possible to get access to data of other installed applet in the card as long as they are installed before our applet even if they belong to another security context. This ability to see data and code of previously installed applets highlights the importance of blocking attackers from installing applets after card issuance. As stated in previous sections, the presence of firewall mechanism will not stop attacks based on misuse of the `getstatic` op-code. With the use of the information retrieved from analysis of the dump result and `putstatic` op-code, we are able to change the code and the data of previously installed applets on the card.

5.4 A novel approach to build type confusion using Java Card API

Type confusion is a technique that uses an unexpected type in order to illegally access the content of an object. In Java Card,

the specification of the API expresses which type has to be used for calling a method. The `arrayCopyNonAtomic` method of the class `Util` which belongs to the package `framework` is of a particular interest. Its signature is as the following: `arrayCopyNonAtomic(byte[] src, short srcOff, byte[] dest, short destOff, short length)`.

It copies the content of source array `src` which is of type byte, into another one of the same type `dest`. The idea is to change the type of the `src` object and to try other types like instances of classes. The type system of Java Card specifies that an array inherits from `object`. In Java Card, each kind of array (byte, short, *etc.*) is on distinct branches of the tree separated from the classes. To the best of our knowledge, this type confusion has never been published.

The virtual machine relies on the BCV to ensure the type correctness of the parameters. Due to our threat model, this verification is not performed. Thus, if there is no BCV and no run time checks, a type confusion concerning the first parameter is possible. If the `src` belongs to another branch of the type system and in particular an instance, it becomes possible to copy the instance into the `APDU` buffer. On a few smart cards, there is a run time check to ensure the type compatibility. We discover that this card does not protect itself against this attack vector. This attack can be added to the attack subtree: Array Extension Attack (AEA) related with array type confusion (not presented here).

5.5 Retrieving the key

As described in the previous section, when a card did not implement a BCV or a run time check to ensure type compatibility, there would be a possibility to get access to the data of one type as another type. In this section, we describe our method to get access to a key object as an array. Due to the fact that there are various ways to read data of an array, we use this type confusion to read value of a key. To perform this attack, we define a `Triple DES` key in an applet and insert the code described in Listing 1 into the applet:

Listing 1 Method used to retrieve key

```
public short CopyObject(byte[]
    dummyArray, DESKey deskey,
     APDU apdu){
Util.arrayCopyNonAtomic(dummyArray, (
    short)0, dummyArray,
     (short)0, (short)16);
apdu.setOutgoing();
apdu.setOutgoingLength((short)(16));
apdu.sendBytesLong(dummyArray,(short)
    0, (short)16);}
```

The corresponding opcode of Listing 1 is changed as shown in the Listing 2. In the modified version, the key is provided to the `arrayCopyNonAtomic` method as source parameter.

Listing 2 Code snippet of `CopyObject` method

```
19 aload_1 -> 1A aload_2
03 sconst_0
19 aload_1
03 sconst_0
... ...
```

We execute the applet which contains the modified code. As the applet is not stopped with an error, we notice that the stack of the platform is not typed and we can continue with the type confusion attack. The data returned from the code might be the key meta-data or the key in an encrypted format. As the platform did not detect any type confusion attack, we check addresses around the reference of the key to see if there is any valuable information there. We need to find reference of the key and to read addresses around this reference as an array. It is shown in Listing 3; the code used to retrieve the key reference.

Listing 3 Method used to get key reference

```
public short getKeyAddress(DESKey
    deskey){
short dummyValue = (byte)0xAA;
return dummyValue;
}
```

The corresponding code of method `getKeyAddress` is represented in Listing 4. Listing 5 represents the malicious modification of Listing 4.

Listing 4 Original code of `getKeyAddress` method

```
Public short getKeyAddress
    (DESKey deskey){
03 //flags: 0 max_stack:1
21 //nargs: 2 max_locals:3
10 AA bspush 0xAA
31    sstore_2
1E    sload_2
78    sreturn
}
```

Listing 5 Modified code of `getKeyAddress` method

```
Public short getKeyAddress
    (DESKey deskey){
03 //flags: 0 max_stack:1
21 //nargs: 2 max_locals:3
10 AA bspush 0xAA
31    sstore_2
19    aload_1
78    sreturn
}
```

With the use of `getKeyAddress` method, we find `0x00B7` as the key reference. Next, we need to use the opcode represented in Listing 6 to read references around this

address. In the corresponding listing, *XX* represents address of the objects that we read as arrays .

Listing 6 Malicious code to retrieve information as array object at specified addresses

```
11 sspush XX
28 astore 04
15 aload 04
03 sconst_0
19 aload_1
03 sconst_0
10 bspush 0x10
8D invokestatic arrayCopyNonAtomic
0B pop
1B aload_3
8B invokevirtua lsetOutgoing
0B pop
1B aload_3
10 bspush 0x10
8D invokevirtual setOutgoingLength
1B aload_3
19 aload_1
03 sconst_0
10 bspush 0x10
8D invokevirtual sendBytesLong
7A return
```

At the address `0x00B8`, we find the key data. At a first glance, we may think it might be the array that we used for key initializing, but after investigation on the reference of the key initializer array, we observe that the reference of this array is different than `0x00B8`. Also, if we change the value of the key initializer array after key initialization; the result shows no change in the value returned as key value. Thus, We conclude that, the gained value belongs to the key value and no other object in the applet.

As in the `buildKey` method of `KeyBuilder` class, the key encryption parameter is set to false, we change this Boolean to `true` and expected to get the key in encrypted format. But the key is always returned in plain format. We conclude that, no secure storage for the keys is implemented in this Java Card Platform.

5.6 Object overflow

Using the `arrayCopyNonAtomic`, we observe that the size of the original source array is not checked. This opens the possibility to break the confidentiality of the data. We experiment this weakness while copying an object into the destination array. We observe a strange behavior of the card. If we copy `This` object into the `APDU` buffer, we get data stored after `This` and in particular the CAP structure as shown in Fig. 2. So the embedded code is visible which allows us to understand how the card manipulates internally the references. Moreover, reversing the elements of the `arrayCopyNonAtomic` method (change the `src` and `dest`) allows us to write from the buffer `APDU` directly into the memory. This leads to store our own code directly into

```
1.  00 C8 00 C9 00 CA 20 00 00 00
2.  1F 06 03 00 00 00 00 C6 00 CB
3.  20 00 00 00 01 06 00 C2 00 00
4.  00 DE 80 00 00 00 00 05 00 0A
5.  73 68 65 6C 6C 63 71 64 65 73
6.  2C 00 00 00 C2 02 00 C3 E0 00
7.  00 00 1E 05 00 01 00 01 06 00
8.  C4 80 00 00 00 00 05 00 01 02
9.  00 00 01 00 00 01 11 11 CA FE
10. 30 1D 78 01 10 04 80 00 00 05
11. 80 00 01 06 80 00 00 07 80 00
12. 00 7A 03 10 AD 01 03 02 38 7A
13. 02 20 19 00 00 78 03 30 00 00
14. 00 00 1E 77 02 40 1E 1F 25 78
15. 03 50 1E 1F 16 04 38 7A 01 12
16. 7C 00 01 30 7C 00 02 31 1D 78
17. 01 42 03 29 04 04 29 05 16 04
18. 78 04 40 18 19 1A 1F 8B 04 0D
19. 3B 04 78 05 42 18 8C 80 6F 18
20. 10 14 90 0B 87 00 18 10 14
```

Fig. 3 A portion of the dump

the card, just with an API call to the adequate function with a single `APDU` command. This adds a new branch into the CFA tree, which will be discussed in Sect. 5.8.

In the Fig. 3, we have used the described method to dump 256 bytes of the `this` object. This applet contains three static fields and two instance fields. We observe that `0x00C8`, `0x00C9` and `0x00CA` are offsets corresponding to these three static fields, and `0x00C6` and `0x00CB` are the instance fields. Normally the size of the `This` object should be limited to the offsets of the instance fields. We can remark that, thanks to the absence of bound checks, we can read in orange the `AID` of the applet. The bytes in red correspond to the header of a method `0x01 0x11`, says flags: 0, max_stack: 1, nargs: 1 and max_locals: 1. Then, in green we can reverse the code of the method: `0x11 0xCA 0xFE 0x30 0x1D 0x78` says `sspush 0xCAFE, sstore_1, sload_1, sreturn`.

More interesting is the way the static fields are managed with the second method: `0x04 0x80 0x00 0x00 0x05 0x80 0x00 0x01` *etc*. We reversed the binary to obtain the following code `sconst_1 putstatic_b 0 sconst_1 putstatic_1` *etc*. With the help of the source code, we understood that the value `0x00 0x00` corresponds to the first index of the static fields says `0x00C8`, and the second to `0x00C9`. This has been confirmed by overwriting this code with another one, using the `arrayCopyNonAtomic` as a write function, reading a field and storing it into another one.

The next step is to understand how the memory is managed within this card. We try to instantiate twice the applet

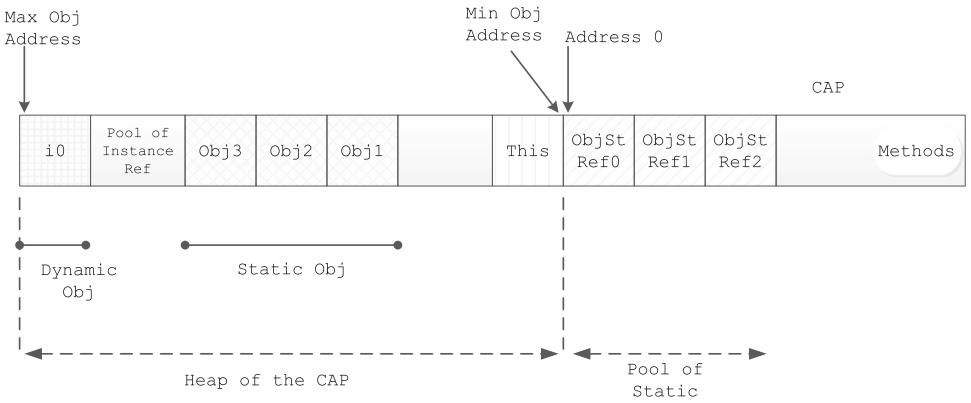

Fig. 4 Description of the memory management of the targeted platform

in the constructor. To understand if the `this` are managed within a linked list or an array. Surprisingly, this card does not support multi instances of `Applet` and returns an error while executing the second instance creation. Thus, the card is not compliant with the standard Java Card 3.0 [24].

We use the dumper/dumped approach to gain access to the data stored before the `CAP` values, as described in Sect. 5.3. We load into the card an applet which has static arrays and one non static array. After dumping the memory, we observe that the memory management is unconventional. The heap is located in the same memory segment of the `CAP`.

Figure 4 illustrates how the memory is managed within this card. Objects are allocated from `MinObjAddress` to `MaxObjAddress`. The first allocated object is the unique `this`. Then, we observe a yet unknown area, then, the static data are allocated from min to max offset. After that, we have the pool of instances, a set of offset followed by the values of the instances. If an instance is dereferenced, then the entry is erased from the pool of Instances, while the value is still present. This seems to be a basic form of garbage collector.

This approach allows a quick uninstall of the applet having in the same segment the heap and the code. We can remark that, this chip has a Memory Management Unit, but the developers did not use this facility or restricted it to the native layers of the card.

5.7 Frame overflow

In the Java Card, each method call leads to a new frame creation. The frames which are stored in the Java Card stack, are temporary data structure which has a set of local variables and an operand stack. Local variables are the variables defined in the current method body. In the area of the local variables, JCVM also stores variables which are passed to the method. The operand stack is the area which the JCVM stores variables related to the current method execution including constants, reference to static fields, reference to objects and also values from local variables area. The sizes of the operand stack and the local variables area are determined and are hard coded into the header of each method at compile time [21]. The size of the operand stack is referred to as `max_stack`.

The Java frame is a non-persistent data structure implemented in different manners and the specification gives no design direction for it [6]. The JCVM should also store return addresses of methods as well as other system data like current context and frame pointer to control the execution flow. Various implementations for storing return address structure can be used. One of the simplest implementations is to store this data in the method frame. Storing the return address data in the method frame has the drawback of more convenient access to this data which leads to the EMAN2 attack. In other Java Card implementations, *Separate Stack* for system and data is introduced. In this way, EMAN 2 attack will not be applicable in such cards.

In this section, we introduce a new attack to get access to the return address in the cards which have implemented a *Separate Stack* countermeasure. This attack is based on the frame overflow and stack underflow. This work is inspired of a previously published countermeasure [12] where the operand stack was split in two parts in such a way we obtain a typed stack.

In our experiments, we want to get access upper the stack boundaries and investigate if the return address is stored above the stack area. In order to do this, we write a recursive method and call it until a frame overflow occurs. If we call the recursive method one time before memory error occurrence (the frame overflow), then we are near to the maximum stack boundaries. Thus, we expect that in some transient area, the return addresses of this chain of recursive methods are stored. We also know that as the return addresses for all of these recursive methods are the same, we should find a pattern of return addresses in an area of memory where an address is iterated. The number of iteration should be the same as the number of the recursive methods call. In the Listing 7 the recursive method is listed.

Listing 7 The recursive method

```
private void exploreFrame(byte
    numberofCalls){
     if(numberofCalls==0)
          return;
     else
          if((numberofCalls==(byte)1))
          {
               //an arbitrary code;
          //we will change it
          //to a malicious code before
              loading
          //the CAP file into the card
          }
          exploreFrame(--numberofCalls)
              ;
}
```

To understand how the card manages its frame and stack boundaries we design some experiments. In all of these experiments there is no on card BCV, so a malicious code can be successfully installed on the card.

5.7.1 Frame overflow and stack overflow

In this experiment, we call a recursive method until we reach to the maximum allowed numbers of frames. Then we push more elements on the stack than the `max_stack` property of the method. This chain of push operations cause to exceed `max_stack` property of the method. Figure 5 represents this operation. The dotted line specifies the boundary of `max_stack` of the current method.

As the result of the experiment, we find that the card checks `max_stack` property of the method and blocks exceeding the `max_stack` property by `0x6F00` error code. On the other side, if the number of push operations do not exceed `max_stack` property, the method is successfully executed. As a conclusion, the stack overflow is blocked in this card. This corresponds to a run time test regarding the top of stack.

5.7.2 Frame collision

Frame collision occurs if a frame which requires more memory than available can be built by an `invoke` op-code. This corresponds to a check while building the frame. We suppose that the kernel stack is above the data stack and launch some experiments. In the targeted Java Card Platform, we could not cause a frame collision because the `max_stack` property of each method is checked by the JCVM. As experiments do not succeed, the check is correctly implemented. So frame collision is also blocked in this card.

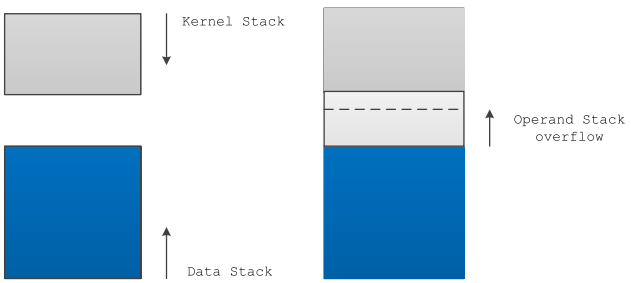

Fig. 5 Frame overflow and stack overflow

5.7.3 A new CFA using frame overflow and stack underflow

In this experiment, we check if we can access upper than stack limits by using `sload` op-code. Thus, when we reach to the maximum allowed frame numbers, we put various index values to `sload` op-code. We call the recursive method in the Listing 7 one time before frame overflow occurrence, and then apply `sload` op-code with various indexes. We iterate this experiment for 256 times and store the result. The resulted data obeys a pattern with some fixed values, which we guess these fix values are return addresses. The number of these fixed values are equal to the number of calling the recursive method. After some experiments, we find a pattern as depicted in the Fig. 6.

Listing 8 The recursive method with an invalid condition

```
private void exploreFrame(byte
    numberofCalls){
     if(numberofCalls==0)
          return;
     else
          if((numberofCalls==(byte)1))
          {
               //an arbitrary code;
           //we will change it
           //to a malicious code before
               loading
           //the CAP file into the card
          }
          exploreFrame(--numberofCalls);
          if(numberofCalls == (byte)(
              MAX_numberofCalls+1))
          //this condition will never be
               satisfied
               ISOException.throwIt((
                   short)0x6234);
}
```

We also change the return address field using `sstore` op-code to see if it causes a change in the program flow. We insert a `throwIt` line in our recursive method, which is listed in the Listing 8. This line of code throws an specified exception. Then we bind this exception to an `if` condition that will never be satisfied. Normally we can never receive the specified exception because the condition is never valid.

In this experiment, we want to check, if changing a byte in the kernel stack that we recognize as the return address, will cause a change in the program flow (i.e. cause to receive the exception at the output). Thus, we change this of the

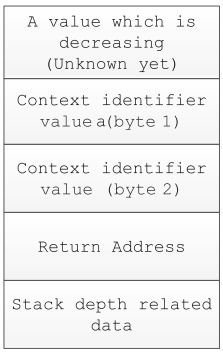

Fig. 6 General pattern for kernel stack

return address to the address of the line of code that throws an exception.

In the kernel stack, the return addresses of the recursive methods corresponds to the `if` condition line. Thus, finding the address of the `throwIt` line was easy. We change the return address in the kernel stack to the address of the `throwIt` line, using `sstore` op-code and then we receive the expected exception at the output. It proves that the recognized byte in the kernel stack, corresponds to the return address of the method and we are able to change the program flow by manipulating the kernel stack's data. The introduced Frame Overflow and Stack Underflow attack allows us to replay old attacks like EMAN2 even in the presence of countermeasure like split kernel and data stack. This shows that the implementation of a security code must be carefully designed and all the issues tackled.

5.8 New branches added to the tree

The `arrayCopy` attack and `frameOverflow` attack are described respectively in Sects. 5.4 and 5.7.3. These two attacks can be added to the previous tree at a higher level. For the `arrayCopy` attack, D4 or D5 or a new one D7 can mitigate this attack. The D7 countermeasure checks the type of the element at `src` and `dest` of the API call. Other API methods must be carefully checked and can suffer the same kind of attack. It must ensure that the elements passed to the API have a compatible type with the API definition. Moreover a secure implementations of the API must ensure that the boundaries of the arrays are correctly verified. It must not be possible to generate a buffer overflow in the memory, as we did in this exploit. Surprisingly, we have successfully performed this attack on several smart cards available on the web stores. Putting `this` as a parameter instead of an array, works on different cards, while the absence of the array bounds checks seems to be specific to this product. The Fig. 7 represents the new branch added to the tree.

The `frameOverflow` attack, which is introduced in Sect. 5.7.3, can be mitigated by two countermeasures: D5 or D8 countermeasures must be implemented to prevent this attack. The D8 countermeasure prevents access beyond the stack boundaries. The D5 countermeasure which is a BCV, may not be present on the cards because of its high cost algorithm.

5.9 Conclusion

In this section, we have described various methods to get information from the internal structure of a Java Card Platform. First, we tried to perform known Control Flow Attacks to redirect program flow to a malicious code stored in an array. As in the targeted Java Card Platform, the gained reference for an array does not directly refer to the array elements, we were not able to perform such attacks. In order to find the reference of our intended array in the card, we have proposed a method based on gaining access to the memory areas using other applets (i.e. the reference of our intended array can not be accessed by the applet which the array is defined within). By the use of this method, we are able to gain access to the area that is inaccessible from within the targeted applet. We can get access to the applets code and data by the use of the `getstatic` op-code. We also can modify code and data of the applets by the use of the `putstatic` op-code even in the presence of the firewall mechanism in the card.

These experiments highlight the importance of implementing countermeasures against malicious use of `getstatic-putstatic` op-codes. By performing exhaustive analysis of the data gained from this attack we have an in-depth understanding of the platform internals.

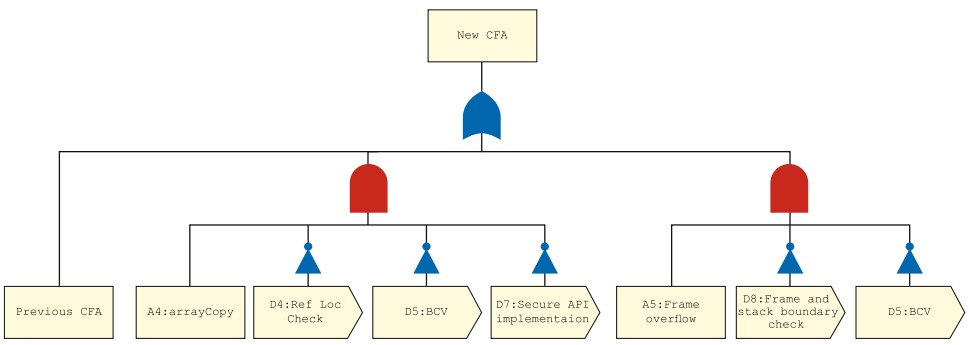

Fig. 7 The new CFA tree

We have also proposed a new method to get access to the applet's object by misusing `arrayCopyNonAtomic` API of the `Util` package. By the use of the proposed method, we are also able to retrieve key data from an installed applet on the card in unencrypted format. It reveals the importance of secure implementation of standard Java Card API and designing secure storage for the applet's keys.

In this section, we have described our new method to get access to the kernel stack of the cards. In the proposed method, we use frame overflow in conjunction with stack underflow to get access to the return addresses of the methods. We have used this method to perform a successful CFA in the cards that are resistant to EMAN2 attack.

6 Countermeasures

This card presents several weaknesses that allow us to retrieve the cryptographic keys in plain text. This has been possible due to the following vulnerabilities which combined provide this successful scenario:

- Ill-typed applet, this can be mitigated using the BCV which is the standard procedure. Nevertheless, this important piece of code must not have any vulnerability like the one pointed out by Faugeron in [15] or the weakness discovered by [20]. It seems reasonable to embed run time countermeasures to mitigate any of the aforementioned vulnerabilities,
- It can be seen, that relying on the BCV is not enough to guarantee the confidentiality of the code segment. We need to clearly inhibit the `getstatic-putstatic` weakness. The pool of static (D6) is of course a good countermeasure which can be implemented with an indirection table but more efficient is a secure implementation of the linker,
- To mitigate the `arrayCopyNonAtomic` weakness, the type of the object to be copied in the array should be correctly verified. This can be obviously done at run time, each object carries its type in its meta data,
- Secure container must be implemented according to the state of the art. When creating a key, it has to be encrypted whatever the Boolean value in `buildKey` method is. The key used for encryption must not be visible to the attacker, even the header is stored sometime some where else (ROM, Crypto processor memory, *etc*),
- Any function that manipulates an array must check the validity of the array bounds(the number of used memory cells).
- The frame management must be correctly implemented (i.e., completely checked) in such a way the stack underflow can not generate a frame overflow.

The right countermeasure against a bad usage of the `getstatic-put-static` is a secure linker. The Java Card specification defines the linking step, which is performed in two step one is external (generation of the token) and one is during the loading of CAP file (resolution of the token). When the software is loaded into the card, the Java Card Virtual Machine provides a way to link the CAP file to install with the installed Java Card API. This step is performed thanks to a token link resolution references in the `Constant Pool` component. To friendly find where each token is used, the `Reference Location` component keeps a list of offsets, in the `Method` Component. The `Reference Location` component makes a link between each token and the `Constant Pool` component. There are two ways to perform the link process.

The linker analyzes linearly the byte array and determines if the parameter has to be linked. If yes it uses the `Constant Pool` component to refer to its internal value and replace it. Such an algorithm has a complexity of $O(n)$, n being the number of byte codes in the byte array. The second option is to use directly the `Reference Location` component which directly provides the offset of the token to be linked. Then, the complexity is $O(p)$, p being the number of tokens in the `Reference Location` component.

The value of n is largely greater than p and most implementations of the linker optimizes the process with the second approach. This raises two issues: if a token to be linked is not referred in the `Reference Location` component, the token is let unchanged (principle of the basic `getStatic` attack). To be functional, i.e. all the tokens must be linked, the linear approach **must** be implemented, but it is not sufficient. Another issue has been pointed out in [16] where the authors use the linker to retrieve information from the card. Another constraint is that, **only** the tokens that follow an instruction requiring a token must be linked. Then, each time a token is resolved, the algorithm has to check that the byte code belongs to the set of 43 instructions requiring a token as parameter. Then, the complexity of the secure algorithm becomes $O(n + 43 * p)$ which of course is more costly than the basic algorithm but it is performed only once. Few implementations respect the first constraint and only one smart card manufacturer has also implemented the second one, leading to a secure linker. Implementing a secure linker completely mitigates the CFA. Other countermeasures have been proposed in the literature [2] and [3] but most of them are related to fault injection attacks. In [22] the authors propose a formal framework for correctly implementing the transactional mechanism.

This study shows that, although it seems obvious that several counter measures have to be implemented, but a correct design of a Java Card Virtual Machine is not an easy task. Several mistakes have been made on this product.

7 Conclusion and future works

In this work, we have presented the evaluation of a development card. We have used a black box approach, discovering step by step the vulnerabilities in the targeted Java Card Platform. At the end, we have been able to get access to the cryptographic keys by combining the different vulnerabilities. Moreover, we have found a means to directly write our own code into the Java Card memory with a single command. This finding offers us the ability to have a sort of an on-line debugger which gives us the ability to completely explore the targeted Java Card. Then we have characterized the memory management understanding the implementation of the pool of statics and the pool of instances. Several parts of the heap are still unknown and need further work to finish the characterization. We can also get access to the kernel stack and the return addresses to perform a successful CFA. As mentioned before, this card had implemented *Separate Stack* countermeasure against CFA, but using the new method introduced in this paper, we can successfully bypass this countermeasure.

In this paper, we have presented two main methods to attack the cards. The first one is Java Card API type confusion attack and the second is the frame overflow attack in conjunction with the stack underflow to get access to the kernel frame.

We have verified that this card is not compliant with the specification because it does not allow multiple instances of the applet.

This card has been announced to have successfully passed European certification at CC level EAL4+. By Reading the certification report, it can be found that only the chip and the cryptographic library have been certified. The Java Card Platform itself has never been part of the TOE.

It is surprising, that such a recent card offers so many vulnerabilities that have been patched for a long time by other smart card manufacturers. Some parts of the Java Card implementation are definitely not at the state of the art. Of course, we have worked on a development card, which can differ slightly from a product. For example, we think that products based on such a card will have the keys encrypted by default, but we are not sure that the check of the array bounds differs in a product.

Although the evaluated Java Card Platform had implemented some recent countermeasures like *Separate Stack*, it was vulnerable to an old and publicly known attack like the basic `getstatic` attack. The targeted Java Card Platform implemented a rather recent version of Java Card specification with support of various cryptographic algorithms, but on the other side we were able to retrieve its applet keys in plain text. The card was implemented the *Separate Stack* countermeasure, while the frame management implementation allowed stack underflow and frame overflow. These results show a weak Java Card which lacks a consistent plan for its security. The results also highlight the importance of using a general threat analysis in designing platforms security to hinder even old attacks.

References

1. Barbu, G., Thiebeauld, H., Guerin, V.: Attacks on Java Card 3.0 combining fault and logical attacks. In: Smart Card Research and Advanced Application, pp. 148–163. Springer, Berlin (2010)
2. Barbu, G., Andouard, P., Giraud, C.: Dynamic fault injection countermeasure. In: Mangard, S. (ed.) Smart Card Research and Advanced Applications, Lecture Notes in Computer Science, vol. 7771, pp. 16–30. Springer, Berlin (2013). doi:10.1007/9783642372889_2
3. Barenghi, A., Breveglieri, L., Koren, I., Pelosi, G., Regazzoni, F.: Countermeasures against fault attacks on software implemented aes: effectiveness and cost. In: Proceedings of the 5th Workshop on Embedded Systems Security, WESS '10, pp. 7:1–7:10. ACM, New York (2010). doi:10.1145/1873548.1873555
4. Bistarelli, S., Fioravanti, F., Peretti, P.: Defense trees for economic evaluation of security investments. In: The First International Conference on Availability, Reliability and Security, 2006. ARES 2006, IEEE (2006)
5. Bouffard, G.: A generic approach for protecting java card smart card against software attacks, Ph.D. thesis, University of Limoges, 123 Avenue Albert Thomas, 87060 LIMOGES CEDEX (2014)
6. Bouffard, G., Lanet, J.-L.: The next smart card nightmare - logical attacks, combined attacks, mutant applications and other funny things. In: Cryptography and Security: From Theory to Applications—Essays Dedicated to Jean-Jacques Quisquater on the Occasion of His 65th Birthday (2012)
7. Bouffard, G., Lanet, J.-L.: The ultimate control fow transfer in a Java based smart card. Comput. Secur. **50**, 3346 (2015). doi:10.1016/j.cose.2015.01.004
8. Bouffard, G., Lackner, M., Lanet, J.-L., Loinig, J.: Heap ... Hop! Heap is also vulnerable. In: Joye, M., Moradi A. (eds.) Smart Card Research and Advanced Applications—13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers, Lecture Notes in Computer Science, vol. 8968, pp. 18–31. Springer, Berlin (2014). doi:10.1007/9783319167633_2
9. Bouissou, M., Bon, J.: A new formalism that combines advantages of faulttrees and markov models: Boolean logic driven markov processes. Rel. Eng. Syst. Saf. **82**(2), 149163 (2003). doi:10.1016/S09518320(03)001431
10. Chen, Z.: Java Card Technology for Smart Cards: architecture and programmer's guide. Addison-Wesley. https://books.google.co.uk/books?id=4WDj4H6pT50C (2000)
11. Common Criteria, Common Criteria for Information Technology Security Evaluation, Part 1: Introduction and General Model (2009) version 3.1, Revision 3 (CCMB-2009-07-001) (2009)
12. Dubreuil, J., Lanet, J.-L., Bouffard, G., Thampi, B.N.: Mitigating type confusion on Java Card. Int. J. Secure Softw. Eng. (IJSSE) **4**(1), 19–39 (2013)
13. El-Idrissi, N.E.J., El-Hajji, S., Lanet, J.-L.: Countermeasures mitigation for designing rich shell code in Java Card. In: Codes, Cryptology, and Information Security - First International Conference, C2SI 2015, Rabat, Morocco, May 26-28, 2015, Proceedings—In Honor of Thierry Berger, pp. 149–161 (2015). doi:10.1007/9783319186818_12
14. Faugeron, E.: Manipulating the frame information with an underflow attack. In: Smart Card Research and Advanced Applications—12th International Conference, CARDIS 2013, Berlin, Germany,

November 27- 29, 2013. Revised Selected Papers, pp. 140–151 (2013). doi:10.1007/9783319083025_10
15. Faugeron, E., Valette, S.: How to hoax an on-card verifier, Accepted Talk at e-Smart, vol. 10 (2010)
16. Hamadouche, S., Bouffard, G., Lanet, J.-L., Dorsemaine, B., Nouhant, B., Magloire, A., Reygnaud, A.: Subverting Byte Code Linker service to characterize Java Card API. In: Seventh Conference on Network and Information Systems Security (SAR-SSI), pp. 75–81 (2012)
17. Hogenboom, J., Mostowski, W.: Full memory read attack on a Java Card. In: 4th Benelux Workshop on Information and System Security Proceedings (WISSEC09) (2009)
18. Hubbers, E., Poll, E.: Transactions and Non-atomic api Calls in Java Card: Specification Ambiguity and Strange Implementation Behaviors. Radboud University Nijmegen, Nijmegen
19. Iguchi-Cartigny, J., Lanet, J.-L.: Developing a Trojan applets in a smart card. J. Comput. Virol. **6**(4), 343–351 (2010). doi:10.1007/s11416-009-0135-3
20. Lancia, J., Bouffard, G.: Java Card virtual machine compromising from a byte code verified applet. In: Smart Card Research and Advanced Applications—14th International Conference, CARDIS 2015, Bochum (2015)
21. Laugier, B., Razafindralambo, T.: Misuse of frame creation to exploit stack underflow attacks on Java Card. In: Smart Card Research and Advanced Applications—14th International Conference, CARDIS 2015, Bochum (2015)
22. Mostowski, W.: Formal development of safe and secure java card applets, Tech. rep. (2005)
23. Mostowski, W., Poll, E.: Malicious code on java card smartcards: attacks and countermeasures. In: Grimaud, G., Standaert, F.-X. (eds.) Smart Card Research and Advanced Applications, Lecture Notes in Computer Science, vol. 5189, p. 116. Springer, Berlin (2008). doi:10.1007/9783540858935_1
24. Oracle, Java Card 3 Platform, Virtual Machine Specification, Classic Edition, no. Version 3.0.4, Oracle, Oracle America, Inc., Redwood City (2011)
25. Roy, A., Kim, D.S., Trivedi, K.S.: Attack countermeasure trees (act): towards unifying the constructs of attack and defense trees. Secur. Commun. Netw. **5**(8), 929–943 (2012)
26. Schneier, B.: Attack trees. Dr. Dobb J. **24**(12), 21–29 (1999)
27. Sun Microsystems, Java Card Platform Security, Technical White Paper, October 2001