Exploiting JCVM on Smart Cards Using Forged References in the API Calls

Sergei Volokitin(✉)

Riscure B.V., Delft, The Netherlands
volokitin@riscure.com

Abstract. This paper presents a novel style of attack which compromises the applet isolation implemented by modern smart cards built on the Java Card platform. System calls (APIs) implemented by all cards tested during our research – from several different manufacturers – fail to perform (sufficient) checks on the ownership of the objects provided by applets, compromising the security of the applet firewall. The practical impact of these vulnerabilities is platform-specific; we show that disclosure of critical private data including secure channel protocol keys is possible on some cards, and that even Secure Elements – with dedicated hardware support for memory isolation – fail to prevent memory disclosure of objects owned by the Java Card Runtime Environment, despite preventing all other known state-of-the-art logical attacks. We demonstrate that physical attacks can also be used to exploit this vulnerability on some smart cards, removing the need for an attacker to first install an applet on the card. Finally, we propose a potential countermeasure for preventing these classes of attacks.

Keywords: Java Card · Logical attacks · Physical attacks · Secure element · Memory isolation · Fault injection

1 Introduction

Being the most widespread smart card platform, Java Card has been a target of numerous studies aimed at the security of all aspects of the platform including logical robustness of the platform and resilience to physical attacks.

One of the distinctive features of the Java Card platform is the support for multiple applets on a single card and an availability of post-issuance of the applets. This feature makes it possible for an attacker, who has the ability to load malicious code on a card, to compromise the security of the platform if logical vulnerabilities are present.

Logical attacks, being extremely cheap to exploit in a scalable fashion, are not simple to apply in the field. In most cases, an attacker does not have card management keys to load malicious code and execute it. Nevertheless, logical attacks can reveal a lot about the architecture of the platform and internal

B. Bilgin and J.-B. Fischer (Eds.): CARDIS 2018, LNCS 11389, pp. 186–199, 2019.
https://doi.org/10.1007/978-3-030-15462-2_13

design, which can be used by an attacker in order to find weak spots of the platform and combine it with other techniques in a successful attack.

Physical attacks, on the other hand, do not require an attacker to be able to load and execute malicious code which makes such attacks much more dangerous. There were a number of papers proposing physical attacks on Java Card platforms [1,10,11] showing their effectiveness, but the limiting factor of physical attacks is the price of the equipment and scalability of the attack [12]. Although the price of physical attacks is traditionally much higher than logical attacks, in recent years availability and price of basic equipment for physical attacks decreased significantly.

Section 2 presents state-of-the-art logical and physical attacks on the Java Card platform. Section 3 discusses the applet firewall and additional countermeasures used to ensure applet isolation on a card as well as common attack techniques used to break applet isolation and its limitations. Section 3.3 introduces a logical attack which uses forged references to break applet isolation using API calls provided by the platform. Section 3.4 reveals results of the evaluation of the logical attack on five different cards from multiple manufacturers. Section 4 presents a physical attack based on a single electromagnetic fault injection allowing an attacker to corrupt code on the card to break applet isolation and read the memory of other applets. Finally, Sect. 5 discusses some of the countermeasures presumably implemented on some of the cards and proposes a few improvements which could make the Java Card platform more secure.

2 Related Work

Various attacks on Java Card platform were published in recent years presenting a number of different ways to perform logical, physical and combined attacks.

The main focus of logical attacks is to break applet isolation assuming that an attacker has or can load malicious code on a card.

The paper of Mostowski and Poll presents a number of logical attacks on the Java Card platform which use ill-formed applets [9]. The authors proposed to break applet isolation by means of type confusion between arrays of different types, arrays and objects and the use of pointer arithmetic to create references to create fake array metadata.

Faugeron proposed a novel attack on an operand stack implementation of a Java Card platform and in particular insufficient checks of `dup_x` instruction, which allows an attacker to copy a number of bytes under the stack bottom which might lead to the disclosure of data belonging to a different context [4]. The proposed attack allowing to read 8 bytes of the stack under the stack bottom was limited to Java Card virtual machine implementation which supports optional in Java Card specification `int` type.

Bouffard and Lanet proposed a number of attacks to break applet isolation and get a memory dump containing code and data of other applets and the runtime environment and even the native code [2]. First proposed technique abuses the `getstatic_b` instruction provided by the virtual machine which lacks

checks on the index to the constant pool. The second attack abused the metadata of a transient array object which had a pointer to the physical memory which could be corrupted in order to read other parts of memory.

Finally, Farhadi and Lanet present an attack which allows reversing internal representation of multiple data structures provided by Java Card virtual machine, key objects in particular, by providing a reference to them in the Java Card API call `arrayCopyNonAtomic` [3]. Authors proposed the use of the API calls to reverse object internal representation but did not attempt to break applet isolation. An attack on applet firewall using API calls is described later in Sect. 3.3.

There were a number of publications discussing physical and combined attacks on the Java Card platform [1,10,11]. The combined attacks on the Java Card platform are used to bypass bytecode verifier present on some of the cards by using a fault injection [1] to corrupt loaded applet or its execution to make it malicious. Physical attacks designed to corrupt the execution flow of an applet [7]. The physical attack described Sect. 4 does not require an attacker to load any code on a card since it relies on a weakness of the platform implementation itself.

With regard to the logical attack approach, there are a number of publications presenting how similar attack method can be applied to a Trusted Execution Environment (TEE). Machiry et al. presented an attack allowing an attacker in control of a user application in Rich Execution Environment to abuse the fact that there are not sufficient checks on the TEE side to leverage the privileged rights of the TEE to escalate privileges in the REE side. As a result, so called confused deputy attack allowed an attacker to break memory isolation.

3 Logical Attack

This section introduces a novel logical attack allowing an attacker to break applet isolation enforced by the firewall. The proposed attack is successful even on some of the most modern and protected Java Card platforms such as Secure Elements which have a memory protection unit for memory isolation. Section 3.1 describes memory allocation and management on most of the modern implementations of Java Card platform. Section 3.2 will discuss some of the published state-of-the-art attacks and countermeasures present on modern cards preventing the attacks. Section 3.3 will present a novel attack which uses Java Card and Global Platform API calls with forged references in order to bypass applet firewall. In this section only the logical attack, which requires an attacker to be able to load and execute code on a Java Card using a set of card management keys, will be discussed. Section 4 will introduce a physical attack which does not require an attacker to be able to execute arbitrary code on a card to read some parts of memory belonging to other applets or the Java Card Runtime Environment.

3.1 Java Card Applet Isolation

Due to the limited resources of typical Java Card cards, applets installed on a card are running in a single Java Card Virtual Machine and share the EEPROM or flash memory for storing data and code. A dedicated feature defined in the Java Card specification, the applet firewall, is required to ensure that an applet cannot access an object owned by another applet. The firewall is designed to ensure that no malicious or erroneous applet can compromise separation between applets and the Java Card virtual machine and get read or write access to other parts of memory.

In order to be able to learn more about the way a Java Card platform is implemented on a card a simple ill-formed applet, as shown in the code fragment below, can be used to get a value of a reference. Such a code cannot be compiled due to type mismatch, but it can be created by manipulating bytecodes of a CAP file.

```
public static short addr( byte[] ptr ) {
    return (short) ptr;
}
```

Such a malicious applet is effective and works on most available Java Card implementations since the operand stack is untyped, and there is no cost-effective way to detect malicious code execution at runtime. The code may be prevented from being loaded onto a card if a full bytecode verifier has been provided on the card. However, this is not the case for most of the implementations on the market.

Execution of the malicious code on a card can reveal information about the way the references are created and handled internally by the virtual machine. Most of the old implementations of Java Card virtual machines have a reference value equal to the physical address in memory where the object is stored. In contrast, most of the modern cards implement a table where the reference value is an index to the table which stores a physical address and, optionally, metadata of an object. The use of an index table allows preventing a lot of attacks published before, such as the creation of fake metadata using pointer arithmetic [2,9,12]. The actual location of the index table in memory and internals depends on the implementation and is not required for the attacks described in this paper.

Despite the fact that it is possible on most of the cards to get the value of a reference and convert it back from short to reference using ill-formed code described below, any use of such a reference will fail with a security exception or with a card mute.

```
public static byte[] ptr( short addr ) {
    return addr; \\ ill-formed, patched after compilation
}

byte[] p1 = ptr(0x0001);   \\ no exception
p1[0] += 1;                \\ security exception or a mute
len = p1.length;           \\ security exception or a mute
```

When access to a content of a reference is attempted, the applet firewall performs checks of the ownership of the object and makes a decision to deny access or not.

3.2 Limitations of State-of-the-Art Attacks

The logical attacks published so far, which rely on type confusion, were mostly focused on type confusion between byte arrays and short arrays owned by the applet [2,9,12]. Such an attack can allow an attacker to break memory isolation by reading twice as many bytes due to the fact that a lot of old implementations would not check an array type and resolve memory address based on the access operation, namely load byte or load short, and an index. Then metadata of the following in memory objects of the applet can be corrupted, leading to reading out or writing to big parts of EEPROM memory, including code and data of other applets. Although this attack was quite successful in the past on older implementations of Java Card virtual machine, modern virtual machines often implement some additional runtime checks which prevent this attack from a successful memory isolation break. There are a number of different ways in which such a countermeasure can be implemented in a virtual machine, for example, checks of a type of an array stored in metadata of an object and the instruction used to access it. A different address resolution of array elements and/or storage of bytes and short array elements using two bytes of memory for each can make the attack useless and in fact, most of the modern solutions implement some of them and as a result, classical type confusion logical attacks do not work anymore.

Another state-of-the-art attack uses lack of checks on `getstatic_<t>` common for a lot of old Java Card virtual machines, which allows an attacker to execute a malicious code with `getstatic_<t>` and incorrect index to the constant pool and read the memory of other applets and JCRE [2]. Such an attack is easy to prevent since a simple runtime check can be added which performs a check of the index to be within constant pool index range.

As it can be seen from modern Java Card implementations, card manufacturers have improved virtual machine implementations and made them more robust against logical attacks. Additionally, there is an increasing number of Java Card platforms which have hardware support of memory isolation, such as Secure Elements with memory protection units. These units, if configured correctly, mean that any logical attack cannot break memory isolation since access to memory is controlled by the hardware. In this section, we will introduce a new logical attack which uses a different approach to break memory isolation of Java Card applets even when additional runtime checks or memory isolation hardware support are present.

3.3 Attack Based on API Calls

The Java Card runtime environment specification states that the Java Card runtime environment is executed as a privileged process which should have

unrestricted access to all the memory of the virtual machine. A common way to exploit a system with kernel and user space separation is to use system calls provided by the kernel which do not have enough checks on the parameters of the system call [6]. This exploitation technique, despite being widely used on traditional systems, has not been applied to Java Card platforms in the research published so far. The implementation of such an attack will be discussed in this section in detail.

There are over one hundred API calls defined in the Java Card API and Global Platform API, and they are fully or partially supported by most smart cards [5,8]. There are API calls providing cryptographic operations, communication, exception support, card management and more. Depending on the parameters and returned types there are different types of API calls. Some API calls do not take references as parameters and they are not vulnerable to the attack proposed in this paper. Some other API calls take a reference or a number of them to applet objects as a parameter and return a data as a result of an operation over the provided data. As an example, the objects of type `javacardx.crypto.Cipher` have a method `doFinal` with the following signature:

```
doFinal(byte[] inBuff, short inOffset, short inLength,
byte[] outBuff, short outOffset)
```

The API call has two references in the parameters, namely `inBuff` and `outBuff`, pointing to byte arrays with input and output data. Finally, there are some API calls which take a reference as a parameter and change a state of the virtual machine or perform some kind of operation. For example, an API call `sendBytesLong` takes a reference and the virtual machine sends the data at the byte array to the terminal.

It is expected that a reference given as a parameter to an API call is to an object owned by the applet or to a global buffer, but in fact, as it was shown above a simple ill-formed method can be used to create a reference which would point to any record in the index table. Some ill-formed code which can be used to read out the memory of other applications is presented in the following listing:

```
public static byte[] getRefBA( short addr ) {
    return addr; // ill-formed code, patched after compilation
}
...
case INS_API_TEST_1:
    bufPtr = Util.getShort(buffer, ISO7816.OFFSET_P1);
    len = Util.getShort(buffer, ISO7816.OFFSET_CDATA);
    apdu.sendBytesLong(getRefBA(bufPtr), (short) 0, len);
    break;
```

The ill-formed code above allows an attacker to request the Java Card virtual machine to send an APDU response with data referenced by `getRefBA(bufPtr)` and offset and length controlled by an attacker. The Java Card virtual machine

has to check that the type of the object is as expected by the API call and that the object at the reference is owned by the applet. In case the check is not present or not complete the content of a byte array owned by another applet or JCRE is returned in the APDU response.

In a similar way, a number of API calls can be used to write to the objects which belong to other contexts. For example, the same API call `doFinal`, exploited to read memory by providing a forged reference of the source buffer, can be used to write to them by using the forged reference for the destination buffer parameter. Since the key used by the method `doFinal` is controlled by the caller an arbitrary write can be achieved by encrypting a buffer and then decrypting the ciphertext to the buffer at the forged reference. Additional, not only Java Card API calls can be used to bypass the applet isolation, for example, most of the Java Cards also support Global Platform specification and there are a number of API calls which can be invoked by an applet, given the privileges, such as `setATRHistBytes` which takes a reference to a byte array, offset and length and sets historical bytes returned as part of the card's ATR.

It is important to note that the proposed attack in this paper, unlike previously published attacks, will work even in the case that there is a memory protection unit on a card. The Java Card runtime environment is supposed to have access to all of the memory regions, including the memory which belongs to JCRE and all the applets, and so insufficient checks of the parameters of the API calls will result in the hardware protection being useless.

3.4 Evaluation

In order to evaluate the applicability and scalability of the attack, it was executed on multiple cards from different manufacturers and the results were analyzed. The Java Card specification does not require a virtual machine to be implemented in a specific way and every manufacturer is free to decide how it needs to work internally and what additional countermeasures are in place. As a result, the applicability of an attack may differ a lot depending on the internals of a virtual machine. The cards used in the evaluation are listed in Table 1.

Table 1. Specification of the cards

Card	Global Platform	Java Card
`card_a_1`	GP 2.1.1	JC 2.2.1
`card_a_2`	GP 2.1.1	JC 2.2.1
`card_b_1`	GP 2.2.1	JC 3.0.4
`card_b_2`	GP 2.1.1	JC 2.2.1
`card_c_1`	GP 2.1.1	JC 2.2.1

The letter in the card name identifies a unique manufacturer, and the number distinguishes between different cards made by the same manufacturer. The cards

with the same letter in the name are made by the same manufacturer – for instance, `card_a_1` and `card_a_2`, may have similar internal implementations since they are produced by the same manufacturer, but are not identical.

Three different API calls of different kinds were used for the evaluation of the attack. First, `apdu.sendBytesLong()` method was tested as a Java Card defined call commonly used in the real-life applets and one of the easiest for an attacker to exploit. Second, `cipher.doFinal()` API call which normally uses a crypto-engine to perform the operation. And finally, an API call defined in the Global Platform specification, `GPSystem.setATRHistBytes()`, which takes a reference to an object and sets the historical bytes sent with an ATR of the card.

The values of references in the Java Card virtual machine implementation grow incrementally from `0x0000`. In our evaluation, the values of all references which are lower than the first object of our test applet (between 0 and the reference of this first object) were tested with the API calls described above. In Table 2 the first number in a cell is a number of successful calls, meaning that the card did not mute or give an exception and returned an expected result. The second number in a cell is the total number of reference tested. Since the goal of the attack is to break isolation of applets, the total number of references tested is different on different cards, since the number of references belonging to other applets is JCRE-specific.

Table 2. The results of the attacks on the cards

Attack	Card				
	`card_a_1`	`card_a_2`	`card_b_1`	`card_b_2`	`card_c_1`
`apdu.sendBytesLong()`	40/139	2/181	38/183	37/125	3/195
`cipher.doFinal()`	40/139	4/181	34/183	37/125	3/195
`GPSystem.setATRHistBytes()`	82/139	136/181	151/183	✗	137/195
SCP keys identified	✓	✓	✓	✗	✓

As can be seen from Table 2, the number of successful calls of `apdu.sendBytesLong()` and `cipher.doFinal()` is close but not identical. This might be explained by similar but not identical checks of the objects provided at the references. In particular, it can be seen that there are some type checks and references to short arrays result in a fail when provided to both calls.

The results for Global Platform API calls differ a lot from the tested Java Card API calls. On one of the cards, namely `card_b_2`, a Global platform API call `GPSystem.setATRHistBytes()` failed for all references including legal values, which indicates rather a functional issue with the API call. For all other cards it was possible to call `GPSystem.setATRHistBytes()` for bigger number of objects in the index table which might be explained by the fact that the checks of the Global Platform calls is implemented in a different way and there are less or no checks on the type and ownership of the object provided.

For all of the cards but one, it was possible to identify the default Secure Channel Protocol keys stored in one of the objects in plain text.

In order to confirm that the objects found on `card_a_1` correspond to the Issuer Security Domain keys and not just a data with the same content, a new key with value `1011...1E1F` was added to the Issuer Security Domain using put key command and the following data was identified at object reference `0x0099`:

```
0x81 0x31 0x80 0x45 0x10 0x11 0x12 0x13 0x14 0x15 0x16 0x17
0x18 0x19 0x1A 0x1B 0x1C 0x1D 0x1E 0x1F
```

It is worth noting that the key send to the card has to be encrypted using the key encryption key to prevent man-in-the-middle attacks as required by the Global Platform specification, but it is stored in plain text in the card.

Applet Class Object

A large number of objects in memory are of substantial length, and so an attacker can read significant parts of the memory of the card containing code and data. It was identified during tests that there is an object preceding the test applet class objects which points to the beginning to the applet instance and contains data objects with metadata and code. An example of such an object of length `0xB0` on the card `card_a_1` is given below:

```
0x01 0x00 0x00 0x88 0x20 0x00 0x00 0x01 0x08 0x00 0x00 0x09
0x00 0x89 0x07 0x08 0x80 0x82 0x00 0x08 0x01 0x02 0x03 0x04
0x05 0x06 0x07 0x08 0x80 0x82 0x00 0x08 0x01 0x02 0x03 0x04
0x05 0x06 0x07 0x08 0xA0 0x82 0x00 0x10 0x00 0x00 0x00 0x11
0x00 0x00 0x00 0x22 0x00 0x00 0x00 0x33 0x00 0x00 0x00 0x44
0x00 0x00 0x00 0x55 0x00 0x00 0x00 0x66 0x00 0x00 0x00 0x77
0x00 0x00 0xFF 0x88 0x80 0x82 ...
```

The test applet code corresponding to the obtained memory dump starts with the following declarations of class variables:

```
static byte[] in =    {0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08};
static byte[] out =   {0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08};
static short[] sbuf = {0x11,0x22,0x33,0x44,0x55,0x66,0x77,0x88};
```

There are a number of class objects at the beginning of the memory chunk and it is clear to see that, apart from the data itself, there is metadata of the objects as well. An attacker can use these objects as the first step for corrupting the metadata of the objects of a malicious, or benevolent, applet. This allows them to read large parts of the memory following the object, as described in previous work [9,12].

4 Physical Attack

The logical attack above successfully breaks applet isolation on multiple modern cards from various manufacturers, despite the presence of software and hardware countermeasures. An attacker with ability to execute arbitrary code on the platform can get read and write access to the objects of other applets and Java Card runtime environment. However, in many cases, an attacker has no means of installing arbitrary code on cards in the field, which means that the logical attack described above is not possible. In this section, we propose a physical attack which relies on the logical weakness of all of the observed Java Card virtual machine implementations, namely insufficient checks of the ownership of objects passed as a parameter to an API call provided by the virtual machine, such as `sendBytesLong()` or `setATRHistBytes()` as described in Sect. 3. This physical attack removes the requirement for an attacker to be able to execute arbitrary code on a card. All the attacker needs is to know – or be able to guess – which command is executing on a card.

For our proof-of-concept example of this physical attack, a reference assignment was chosen as a target for a fault injection. In practice, there are many more places where a successful glitch can result in a corruption of a reference value. Below, we provide an example of the type of code which could be targeted by this attack:

```
byte[] ref1;
byte[] ref2 = {...}
...
ref1 = ref2;  //<- GLITCH HERE to corrupt a reference assignment

res = anyApiCall(ref1);
send(res);
```

Code which can be targeted using this physical attack needs to have an operation on a reference – an assignment in this case – which can be corrupted using fault injection, and an API call of Java Card or Global Platform which uses the reference. Such code is standard for a Java Card implementation and can be found in virtually all applets.

Evaluation

We performed an evaluation of the effectiveness of this attack on `card_a_1`. We used electro-magnetic fault injection, since the card has a voltage sensor. The card was not decapped, which increases the probability of an attack being successful because it is less likely to damage the card while decapping.

The code which we used for the evaluation of the physical attack is shown in the listing below:

```
byte[] ref1;
byte[] ref2 = {}

ref1 = ref2;
ref2 = ref1;
 <200 times to ease timing>  // <- GLITCH HERE
ref1 = ref2;
ref2 = ref1;

apdu.setOutgoing();
apdu.setOutgoingLength(len);
apdu.sendBytesLong(ref2, offs, (byte) len);
```

An attacker performing fault injection attack has little control over the value of a reference after a fault introduced in the assignment and in most cases the reference value will be arbitrary and the API call `sendBytesLong()` returns the content at the reference back in the APDU.

We performed a fine-grained scan of the chip. In total, we attempted the attack two million times, with a success rate of around 1%. The results of this scan, with the successful glitches marked as red dots, are shown in Fig. 1.

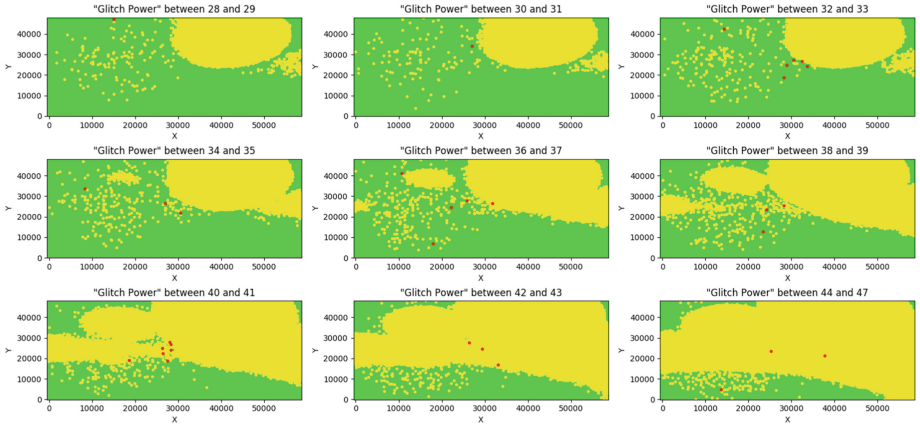

Fig. 1. EMFI results on `card_a_1` (Color figure online)

The initial value of the reference before corruption was equal to `0x0091`. As a result of the test, the reference was corrupted to 13 unique values, providing a different delay, location, and length of the glitch. The obtained values of the reference are shown below:

```
0x0000, 0x0001, 0x0002, 0x0004, 0x0006, 0x0014, 0x0018,
0x0019, 0x001A, 0x0089, 0x0094, 0x009C, 0x00A0
```

Obviously, not all these corruptions could be used by an attacker to get access to other objects in memory of a card. For example, the most common fault injection corruption outcome is `0x0000` which will result in Null Pointer Exception when passed to the API call. On the other hand, some of the values correspond to the objects of other applets and JCRE itself.

As shown in Table 2 for `card_a_1` and `setOutgingAndSend()` API call, there were 40 unique references which could be used with this API call on this card. This means that if the value of a reference is corrupted to any of these identified values, an attacker can break applet firewall isolation and read data of another applet on the card with a single glitch and no malicious code running on the card. In fact, one of the values of the corrupted reference is equal to `0x001A` which corresponds to an object of length 184 bytes apparently containing an applet AID along with both code and data. This physical attack allowed us to successfully obtain the contents of this object, with potentially serious impact since the RID of an applet belongs to a bank.

The physical attack introduced in this section serves as a proof-of-concept of an attack which relies on a logical weakness in the way modern Java Card virtual machines are handling and checking the parameters provided in the API calls. This physical attack shows how an attacker can exploit the vulnerability of a virtual machine using single fault injection to break applet isolation. Although it is difficult to control the address of a pointer after a successful glitch, a number of attempts can allow an attacker to read memory of other applets and Java Card Runtime environment and in some cases, if the corrupted value corresponds to the Secure Channel Protocol keys as it was shown for the logical attack, potentially, can reveal the key values and get full control over a card. Although the fine-grained scan above was performed using two million attempts, a real-world attacker would be able to choose optimized parameters and obtain success with far fewer attempts.

5 Conclusions

In the past years, the security of Java Card implementations has greatly improved. Many vulnerabilities allowing logical attacks have been fixed, and and as a result, there are a number of Java Card virtual machine implementations which are not vulnerable to type confusion attacks. The usage of index tables was one of the countermeasures which lead to a number of logical attacks becoming impossible.

The novel attack proposed in this paper introduces a way to exploit the API calls provided by the platform. The evaluation of the attack on multiple cards from different manufacturers revealed that all of them are vulnerable to some extent to this attack, including the most protected implementations with hardware support for memory isolation.

Additionally, the evaluation of the attack with different API calls showed that the way parameters are checked is not consistent and in most cases the checks on the parameters in the Global Platform API is much less strict and a bigger number of objects of other applets can be read and modified. On all but one card it was possible to identify the objects containing the Secure Channel Protocol keys stored in plain text. Finally, one of the objects identified contains the beginning of the test applet class file with data and metadata of class owned objects which allow an attacker to corrupt metadata of objects of the malicious applet and read and modify most of the card memory.

Logical attacks prove to be useful to identify weaknesses of the platform, but they are difficult to use in real life because in many cases an attacker has no means of loading and executing malicious code on a card. The physical attack proposed in this paper shows a way an attacker can exploit a weakness in the platform using single fault injection to corrupt a reference value used in an API call and as a result bypass the applet firewall.

The Java Card virtual machine specification requires the applet firewall to perform checks of the objects when access to the objects is performed using one of the virtual machine memory access bytecodes, meaning that all of the Java Card virtual machines already have ownership checks implemented and all of the objects in a virtual machine already have to have labels indicating an owner of the object which makes implementation of the countermeasures a trivial task. Having consistent checks of the reference parameters provided in the API calls would solve the issue and make it impossible for attackers to bypass applet firewall using such an attack.

References

1. Barbu, G., Duc, G., Hoogvorst, P.: Java card operand stack: fault attacks, combined attacks and countermeasures. In: Prouff, E. (ed.) CARDIS 2011. LNCS, vol. 7079, pp. 297–313. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-27257-8_19
2. Bouffard, G., Lanet, J.-L.: Reversing the operating system of a Java based smart card. J. Comput. Virol. Hacking Tech. **10**(4), 239–253 (2014)
3. Farhadi, M., Lanet, J.-L.: Chronicle of a Java Card death. J. Comput. Virol. Hacking Tech. (2), 1–15 (2017)
4. Faugeron, E.: Manipulating the frame information with an underflow attack. In: Francillon, A., Rohatgi, P. (eds.) CARDIS 2013. LNCS, vol. 8419, pp. 140–151. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08302-5_10
5. GlobalPlatform. GlobalPlatform Card Specification 2.2, March 2006. http://www.win.tue.nl/pinpasjc/docs/GPCardSpec_v2.2.pdf

6. Machiry, A., et al.: Boomerang: exploiting the semantic gap in trusted execution environments. In: Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS) (2017)
7. Mesbah, A., Mezghiche, M., Lanet, J.-L.: Persistent fault injection attack from white-box to black-box. In: 2017 5th International Conference on Electrical Engineering - Boumerdes (ICEE-B), pp. 1–6 (2017)
8. Sun Microsystems. The Java Card application programming interface (API), 3.0.5, October 2015. https://docs.oracle.com/javacard/3.0.5/api/index.html
9. Mostowski, W., Poll, E.: Malicious code on java card smartcards: attacks and countermeasures. In: Grimaud, G., Standaert, F.-X. (eds.) CARDIS 2008. LNCS, vol. 5189, pp. 1–16. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85893-5_1
10. Rothbart, K., Neffe, U., Steger, C., Weiss, R., Rieger, E., Mühlberger, A.: Power consumption profile analysis for security attack simulation in smart cards at high abstraction level. In: Proceedings of the 5th ACM International Conference on Embedded Software, pp. 214–217 (2005)
11. 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). https://doi.org/10.1007/978-3-540-72354-7_12
12. Witteman, M.: Java Card security. Inf. Secur. Bull. **8**, 291–298 (2003)