Full Memory Read Attack on a Java Card

Jip Hogenboom and Wojciech Mostowski

Department of Computing Science
Radboud University Nijmegen, The Netherlands
j.hogenboom@student.ru.nl, woj@cs.ru.nl

Abstract. We present a simple attack on a Java Card smart card to perform arbitrary memory reads. The attack utilises a known technique of type confusion of the card's Java Virtual Machine by exploiting the faulty transaction mechanism implementation. The type confusion attack lets us access the application's private meta-data, reverse engineer it, and in turn get full read and write access to arbitrary memory locations on the card. The attack gives us good insights into overall memory organisation of the card. We discuss the exploit in detail, including the exploit applet source code, to provide a reproducible attack. We shortly discuss the usefulness of an on-card Bytecode Verifier, that the exploited card is equipped with, and also the Java Card firewall mechanism deficiencies.

Keywords: Java Card, memory attack, type confusion, transactions

1 Introduction

Smart card security is always considered to be very high priority due to the nature of smart card products. Smart cards store our private data (e.g. fingerprints in the new generation e-Passports), allow access to our bank account (electronic cash applications), or protect access to buildings and alike, just to name a few applications. The security of smart cards is considered on many levels, from the hardware layer to application software level. Some well known attacks related to the hardware level are power analysis attacks [4] or fault injection attacks [1]. In general, power analysis uses the information about power consumption of a smart card to reconstruct some secret information from the card (e.g. a private key). Fault injection attacks are attacks where cards are subjected to physical stress or manipulation to cause hardware malfunction which in turn possibly causes the card to behave in an unintended way allowing a backdoor entrance. Given the current advances in smart card technology, both mentioned kinds of attacks are very difficult to perform and require very expensive equipment as well as considerable expertise.

The other kind of attacks, one of which we are about to present in this paper, are software or logical attacks. The underlying assumption is that there exists an exploitable bug in the application running on the card or the card's operating system. Similarly to desktop computers certain kinds of such bugs can be exploited on smart cards: access or change of the private application's data

might be possible or execution of the attacker's code (like in the well known buffer overflow attacks).

Building on our previous work [7] we present a concrete attack on one particular Java Card [9, 8]. A Java Card is a multi-application smart card that is equipped with a limited version of the Java Virtual Machine and runs Java bytecode. Our attack exploits deficiencies in the Java Card Virtual Machine implementation to result in reading or writing most of the card's memory. The consequence is the possibility to read and modify other applet's data and code on the card.

We should note here that this attack, although very powerful in its nature (we would even call it "comprehensive", i.e. giving access to the whole card) is only harmful to open cards. By this we mean that only cards that still have the possibility to install new applets are susceptible to this kind of attack. Most of the cards in the field are closed for applet installation. Thus, this attack is not a real threat to cards currently deployed. However, it proves the necessity for very strong protection mechanisms on Java Cards. The card that we successfully exploited is equipped with an on-card bytecode verifier (BCV). The bytecode verifier should have protected this card from this kind of attack in the first place. A bytecode verifier protects against invalid bytecode only during bytecode installation time. In our attack the bytecode running on the card is "affected" after installation time, by exploiting another loophole in the JCVM implementation on the card: a bug in the transaction mechanism. In contrast to bytecode verification, Java Card also offers a run-time protection for applet data by the means of the Java Card firewall. Because of its weak design, the firewall mechanism also fails to protect the card.

Finally, although the particular card that we exploited is an older generation Java Card (version 2.1.1) and its various problems are well known, a quick Google search shows that some versions of this card are still available on-line for purchasing. However, we do not and cannot claim that these cards are exactly the same as the one that we used.

1.1 Related Work

Software attacks on Java Cards are now well described in the literature. In [10] Marc Witteman gives general principles of such attacks and gives some examples of pre-installation bytecode manipulation to exploit Java Cards, as well as providing the idea of using the firewall mechanism to achieve similar results to bytecode manipulation, namely introducing type confusion in the JCVM. Based on these ideas we tried various techniques to introduce type confusion on several concrete Java Cards and described the results and behaviour of the protection mechanisms implemented on these cards [7]. In the present paper we analyse in detail and further exploit one of the results from [7].

There is also an attack that gives similar results to ours based on the exploitation of the `getstatic` bytecode instruction and pre-installation bytecode manipulation [3, 2]. Because the `getstatic` attack partly relies on pre-installation bytecode manipulation it would not easily get past the BCV. The authors of [3]

claim possible workarounds for the BCV cards, however this makes the attack even more complex. Our attack, on the other hand, is very simple and does not require any direct bytecode manipulation at all.

1.2 Outline

The rest of this paper is organised as follows. Section 2 describes the building blocks for our attack. Section 3 describes the actual attack and Section 4 briefly discusses the code of the applet to perform the attack. Section 5 describes the results of the attack on the card that we used. Finally Section 6 shortly discusses the protection mechanisms of Java Card and summarises the paper.

2 Building Blocks

The main ingredient of our attack is getting access to an application's private meta-data. This meta-data holds the details about allocated arrays: their types, sizes, and memory pointers. Changing this meta-data allows us to make otherwise unauthorised, arbitrary memory reads and writes from the exploit applet. To get access to the applet meta-data we introduce type confusion by exploiting a faulty transaction mechanism implementation [7]. Section 2.1 explains what type confusion is and how we used it, and Section 2.2 explains how to exploit the transaction mechanism to introduce type confusion into the running bytecode.

2.1 Type Confusion

To introduce type confusion into the Java Virtual Machine running on the card means to trick the JVM to create two references of different types pointing to the same memory location (object on the heap). In our attack the purpose is to hold a byte array reference and a short array reference to the same memory block. Assuming that the memory block was originally (legally) allocated as a byte array, accessing it as a short array gives the possibility to read the data beyond the legal byte array boundaries. Limited by the array size, reading a given number of elements from the short array enables us to access a memory block twice the size of the original byte array. The second half of the short array is the normally inaccessible memory area beyond the byte array. Figure 1 depicts this.

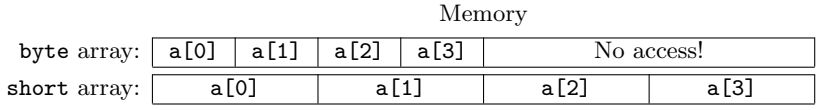

Fig. 1. Type confusion on arrays

When the size of the original byte array is large enough, accessing the block of memory beyond it will give us access to other data on the card. In general, this data might be anything, depending on the memory organisation of the card. For example, we may reach into the memory area reserved for another applet, or to the memory area where the operating system or the JVM keeps its meta-data associated with the given application.

The most straightforward way of introducing such type confusion is to edit the bytecode to be installed on the card so that it is not type correct. The type confusion explained above requires only minor bytecode modification to the code produced by the Java compiler [10, 7]. However, the problem is that such modified, hostile code will not be accepted by a card equipped with a BCV – the card will detect that the bytecode is not well typed and refuse applet installation. Our card does have a BCV, hence we need another way to introduce the type confusion. A bug in the transaction mechanism implementation allows us to do so.

2.2 Transaction Mechanism Bug

The transaction mechanism of Java Card allows the programmer to protect the consistency of the persistent data stored on the card in the EEPROM memory. The main applet class fields are notable examples of persistent data that is stored in EEPROM. The other type of memory that Java Card utilises is RAM for all local variables and temporary computations. The transaction mechanism is only concerned with the contents of the EEPROM memory, the RAM memory is not affected by transactions. The access to the transaction mechanism is provided by the following Java Card API methods [9]:

- `JCSystem.beginTransaction()` – begins an atomic transaction,
- `JCSystem.commitTransaction()` – commits an atomic transaction, all updates to persistent memory since `beginTransaction()` are executed in one atomic step,
- `JCSystem.abortTransaction()` – aborts the transaction, all updates to persistent memory since `beginTransaction()` are ignored. An implicit abort may also be caused by an abnormally terminated program run (e.g. card power loss on card tear).

In Java Card objects allocated with `new` are placed in EEPROM. Following the transaction rules for persistent memory, any possible object allocation inside a transaction should undergo a transaction rollback on abort. The Java Card language specification [9, Section 7.6.3] states that in this case references to all objects allocated within the scope of a transaction should be set to the `null` value and the memory allocated by those objects can be reclaimed. This is where a possible transaction mechanism bug[1] can be exploited. In the buggy transaction

[1] In fact, this bug was so common in early Java Card implementations that later specifications allowed the implementations to prohibit object allocation inside transactions altogether.

mechanism only the references kept in persistent variables (class fields) are reset to `null`. References kept in local, transient variables are left untouched upon transaction abort. These untouched references are dangling pointers to memory areas freed by a transaction abort. Subsequent memory allocation will reuse the reference and if this new allocation has a different type than the aborted one, type confusion occurs. The following code snippet illustrates this idea:

```
short[] arrayS; // instance field, persistent
byte[] arrayB; // instance field, persistent
...
short[] arraySlocal = null; // local variable, transient
JCSystem.beginTransaction();
  arrayS = new short[1]; // allocation to be rolled back
  arraySlocal = arrayS;  // save the reference in local variable
JCSystem.abortTransaction(); // arrayS is reset to null,
                             // but not arraySlocal
arrayB = new byte[10]; // arrayB gets the same reference as arrayS
                       // used to have, this can be tested:
if((Object)arrayB == (Object)arraySlocal) ... // this condition is true
```

At the end of execution of this piece of code we have two variables of different types (a byte array and a short array) with the same reference.

Note that this is fully legal Java Card code and no modification of the byte-code produced by the compiler has been made. Thus, this program gets past the BCV and can be installed on our card.

3 The Attack

We now have all the techniques that we need to craft our attack. As mentioned already, our purpose is to read the memory area of our applet where (we hope) the meta-data is kept, especially the information about allocated arrays. Then by changing this meta-data we can try to exploit the card.

To read the applet meta-data we had to allocate a sufficiently large byte array and, by using the type confusion technique, read the memory block that follows the allocated array. This indeed gave us meaningful results, i.e. the data that we read contained array allocation information. Educated experimentation (allocation of various arrays) and analysis of this data revealed how the array meta-data is stored in the applet's memory. For every allocated byte array we found a sequence of 5 bytes, like this one:

```
0B80FFA54C
```

The first byte indicates the type of the array element (`0B` defines a byte array). The next two bytes `80FF`, excluding the first bit, store the length of the array, in this case 255. The last two bytes `A54C` contain the actual pointer to the array data.

At this point nothing prevents us from changing this meta-data to see if it would really have impact on the corresponding byte array instance variable in the

applet. Indeed we could change the size of the array without any problem, as well as the pointer value, and call the Java code to read the modified array reference. By going through the whole range of possible pointers we can, in theory, read (and write) arbitrary memory locations on the card. As it turned out, not all pointer values resulted in a successful read, we discuss this in Section 5.

4 The Exploit Applet

The full source code of the exploit applet can be found in the appendix. In this section we discuss its general structure.

During the attack preparation phase type confusion is introduced by the transaction mechanism bug following the schema from Section 2.2. After this there are two persistent variables in the applet called `arrayB` and `arrayS` pointing to the same memory block. The corresponding types of these variables are byte array and short array.

Then the applet allocates one special array to target our attack. Its reference is stored in the `arrayMutable` variable. Later, after the attack is prepared (type confusion is enforced and the applet meta-data can be accessed), we change this meta-data to make the `arrayMutable` variable point to arbitrary memory locations. Initially the length of `arrayMutable` is set to some predefined value, so that we can easily find its corresponding meta-data. If we set the length to 2, we have to find the meta-data byte sequence `0B8002`. Once the corresponding meta-data is found, its location is permanently stored in the applet in the `index` variable.

At this point the attack is fully prepared, the applet is now ready to read arbitrary memory blocks. This is done with a separate APDU command that carries the length and the pointer value to be injected into the `arrayMutable` reference. After this the contents of the array, and hence the requested memory block at the given pointer, can be read and sent back to the host.

5 Attack Result

At this point it is up to the host/terminal application to call the applet with the whole range of pointer values to read out the card's memory block by block. We have written a small host application to do just that. Running it on our test card allowed us to read around the total of 48KB of data out of the card in continuous blocks of various sizes starting at different offsets.

Not all of the addresses in the range of `0000` to `FFFF` were readable, see Figure 2. This was to be expected. The card specification declares the size of the card's EEPROM to be 32KB. Since we managed to read more than this we suspect that parts of the card's RAM and/or ROM are also mapped into the `0000`–`FFFF` address range. We identified the complete code and data of all the applets installed on the card, including our exploit applet. We did not yet identify the rest of the data that we read out. We *suspect* that the blocks that we read map to the Global Platform card manager, the operating system of the card, or

Address range	(Suspected) contents
0020–0F20	system data
3017–3717	RAM, system data
4000–FFFF	applet data and code, GP card manager

Fig. 2. Memory organisation of the exploited card.

the RAM, see Figure 2. The exact meaning remains to be analysed. However, we do consider the card fully compromised as we could read and modify out all the code and data belonging to all other applets.

6 Discussion

We have presented a relatively simple software attack to fully exploit one particular type of Java Card. The main reasons why this particular attack scenario worked are the following:

- The card has a buggy transaction mechanism that allowed us to introduce type confusion into the card's JVM despite the presence of the on-card BCV.
- Then, we discovered that the applet's meta-data is kept at the end of the memory area allocated for that applet.
- Finally, we managed to analyse the meaning of the meta-data, this in turn allowed us to change it to make arbitrary memory reads.

A card with a different memory organisation would have been more difficult to attack, but we think a similar attack should still be possible, provided that the card would be susceptible to type confusion (either by the transaction mechanism bug or direct bytecode manipulation). For example, in case the meta-data would be stored at the beginning of the applet's memory area, instead of at the end, we would simply have to craft and install two applets. The first applet would perform the same type confusion attack as before to reach into the second applet's meta-data to change the array reference. From this point on the attack would proceed in the same way as the one we presented.

The point for discussion is the adequacy of the protection mechanisms on the Java Card we exploited. In general the card provides two protection mechanism: the on-card bytecode verifier and Java Card firewall.

We already discussed the purpose of bytecode verification. The bytecode verifier has been proved insufficient for the kind of attack that we presented. This is because the Java code that we run on the card is fully legal from the bytecode verifier point of view.

The firewall mechanism is designed to keep applets in their sand boxes and separate their data. This should provide another line of defence for the card and applets installed on it. However, the generic problem with the firewall is that it protects data by the Java reference and not by the actual memory pointer [9, 7]. I.e. it is the access to the reference that is limited to a given applet/sand box and not the memory block that this reference points to. When the reference is

legal (belonging to the applet accessing it), but the pointer is not, the firewall does not detect the unauthorised memory access. A more elaborate Unix-style segmentation control would be required here, but it would affect performance and it is not considered by the Java Card standard [9]. On top of that there are other known deficiencies in the Java Card firewall mechanism [3, 5, 6, 2].

Because of the insufficient design and bugs in the implementation of the operating system of the card, the two mentioned mechanisms failed to protect the card against our attack. In contrast, we were not able to exploit any other card, from the eight different ones that we have [7], in a similar way. Only one other card out of this eight is equipped with a bytecode verifier. All of those cards implement the obligatory firewall mechanism, but this is flawed by design in the way we just described. The main reason for these other cards to resist our attacks are all kinds of additional protection mechanisms. In particular, they employ run-time checks [7] to dynamically prevent type confusion. The exact workings of such (usually not documented) run-time checks is very difficult to examine on Java Cards, and they can range from simple integrity checks on references to full run-time type checking. The bottom line is that run-time mechanisms can provide better protection than one time, pre-installation check. We are not claiming that a bytecode verifier is in general obsolete (it did prevent us from exploiting one other card), rather that it should be accompanied by other mechanisms.

Finally, we would like to state again, that we do not consider this attack a serious threat for card applications currently in use. Such cards do not have the possibility to install new applets and thus should be resistant to our attack. However, the analysis of the extracted data, especially the part which that we believe belongs to the card manager or operating system, may give us possibilities to further exploit the card.

Acknowledgements We thank Erik Poll for his valuable comments.

References

1. Sudhakar Govindavajhala and Andrew W. Appel. Using memory errors to attack a virtual machine. In *IEEE Symposium on Security and Privacy*, pages 154–165, 2003.
2. Konstantin Hyppönen. Use of cryptographic codes for bytecode verification in smart card environment. Master's thesis, University of Kuopio, Department of Computer Science, June 2003. Available at `http://www.cs.uku.fi/~khiouppe/masters.pdf`.
3. Julien Iguchi-Cartigny and Jean-Louis Lanet. Developing a Trojan applets in a smart card. *Journal in Computer Virology*, September 2009.
4. Adam Matthews. Low cost attacks on smart cards: the electromagnetic side-channel. Technical report, NGSSoftware Insight Security Research, 2006. Available at `http://www.ngssoftware.com/research/papers/EMA.pdf`.
5. Michael Montgomery and Ksheerabdhi Krishna. Secure object sharing in Java Card. In *Proceedings of the USENIX Workshop on Smartcard Technology (Smartcard '1999), Chicago, Illinois, USA, May 10–11*, 1999.

6. Wojciech Mostowski and Erik Poll. Testing the Java Card Applet Firewall. Technical Report ICIS–R07029, Radboud University Nijmegen, December 2007. Available at `https://pms.cs.ru.nl/iris-diglib/src/icis_tech_reports.php`.
7. Wojciech Mostowski and Erik Poll. Malicious code on Java Card smartcards: Attacks and countermeasures. In *Smart Card Research and Advanced Application Conference CARDIS 2008, Proceedings, Egham, U.K.*, volume 5189 of *LNCS*, pages 1–16. Springer, September 2008.
8. Sun Microsystems, Inc., `http://www.sun.com`. *Java Card 2.2.2 Application Programming Interface Specification*, March 2006.
9. Sun Microsystems, Inc., `http://www.sun.com`. *Java Card 2.2.2 Runtime Environment Specification*, March 2006.
10. Marc Witteman. Java Card security. *Information Security Bulletin*, 8:291–298, October 2003.

A The Exploit Applet Code

```
import javacard.framework.*;

public class DumpMemApplet extends Applet {

  private static final byte
    INS_PREPARE1 = 0x01, INS_PREPARE2 = 0x02, INS_READMEM = 0x04;

  private static final short TEST_LEN = 128;

  private short[] arrayS;
  private byte[] arrayB;
  private byte[] arrayMutable;
  private short index;

  public static void install(byte[] array, short off, byte len) {
      new DumpMemApplet().register();
  }

  public void process(APDU apdu) {

    if (selectingApplet()) { return; }

    byte[] buf = apdu.getBuffer();
    switch (buf[ISO7816.OFFSET_INS]) {
      case INS_PREPARE1:
        arrayMutable = new byte[2];
        short[] arraySlocal = null;
        JCSystem.beginTransaction();
          arrayS = new short[1];
          arraySlocal = arrayS;
        JCSystem.abortTransaction();
        arrayB = new byte[TEST_LEN];
        arrayS = arraySlocal;
```

```
        if((Object)arrayS == (Object)arrayB) {
          Util.setShort(buf, (short)0, (short)arrayB.length);
          Util.setShort(buf, (short)2, (short)arrayS.length);
          apdu.setOutgoingAndSend((short)0, (short)4);
        }else{
          ISOException.throwIt(ISO7816.SW_WRONG_DATA);
        }
        break;
      case INS_PREPARE2:
        copyArray(true);
        index = findIndex();
        Util.setShort(buf, (short)0, index);
        apdu.setOutgoingAndSend((short)0, (short)2);
        break;
      case INS_READMEM:
        byte p1 = buf[ISO7816.OFFSET_P1];
        byte p2 = buf[ISO7816.OFFSET_P2];
        apdu.setIncomingAndReceive();
        short len = Util.getShort(buf, ISO7816.OFFSET_CDATA);
        setupArray(index, p1, p2, len);
        copyArray(false);
        Util.arrayCopyNonAtomic(arrayMutable,(short)0,buf,(short)0,len);
        apdu.setOutgoingAndSend((short)0, len);
        break;
      default:
        ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
    }
  }

  private void copyArray(boolean from) {
    short half = (short)(arrayS.length / 2);
    for(short i=0; i<half; i++) {
      if(from) {
        Util.setShort(arrayB, (short)(i*2), arrayS[(short)(half+i)]);
      } else {
        arrayS[(short)(half+i)] = Util.getShort(arrayB, (short)(i*2));
      }
    }
  }

  private short findIndex() {
    for(short i=0;i<arrayB.length;i++) {
      if(arrayB[i] == (byte)0x0B &&
        arrayB[(short)(i+1)] == (byte)0x80 &&
        arrayB[(short)(i+2)] == (byte)0x02) {
        return i;
      }
    }
    return -1;
  }
```

```
  private void setupArray(short index, byte p1, byte p2, short length) {
    Util.setShort(arrayB, (short)(index+1), length);
    arrayB[(short)(index+1)] |= (byte)0x80;
    arrayB[(short)(index+3)] = p1;
    arrayB[(short)(index+4)] = p2;
  }
}
```