Logical attacks on secured containers of the Java Card platform

Sergei Volokitin[1] and Erik Poll[2]

[1] Riscure B.V., The Netherlands
volokitin@riscure.com
[2] Digital Security group, Radboud University Nijmegen, The Netherlands
erikpoll@cs.ru.nl

Abstract. The Java Card platform provides programmers with API classes that act as containers for cryptographic keys and PIN codes. This paper presents a first systematic evaluation of the security that these containers provide against logical attacks, for a number of cards from different manufacturers.
Most cards we investigated do not appear to implement any integrity and confidentiality protection for these containers. For the cards that do, this paper presents new logical attacks that bypass these security measures. In particular, we show that the encryption of keys and PINs by the platform can be defeated using decryption functionality that the platform itself offers, so that logical attacks can still retrieve plaintext keys and PINs.
We also investigate the possibilities for type confusion to access the global APDU buffer and the presence of undocumented bytecode instructions.

1 Introduction

Like any smart card, a Java Card smart card can be subjected to logical attacks that exploit software flaws or to side-channel attacks, such as power analysis, glitching or other forms of fault injection [12, 15, 1]. In addition to the usual security risks posed by the possibly hostile environment, there is the risk of potentially malicious Java Card code on the card itself. Many Java Card cards do not have an on-card bytecode verifier, due to the resource constraints, which allows malformed code to be installed. Such code can break the security guarantees of the Java(Card) platform, notably memory-safety. Even on cards with a bytecode verifier, well-formed bytecode may still exhibit unwanted behaviour because of bugs in the platform implementation (e.g. in the transaction mechanism [11]) or through fault injection [2].

Some cards do runtime checks as a defence against malformed code. In principle, doing lots of runtime checks – effectively doing full runtime type checking – would prevent all ill-formed code from executing, but the overhead is prohibitively high.

The Java Card API provides several classes for security-sensitive data, notably the subclasses of `Key` for storing cryptographic keys and the `OwnerPIN`

class for PIN objects. The platform could – and maybe should – implement security measures for these objects against logical and/or physical attacks, to protect integrity and confidentiality of keys and PINs and integrity of PIN try counters. Indeed, applications may deliberately store data in `Key` objects, even though the data is not a key, just to benefit from any platform-level protection of these containers [16].

This paper investigates the security measures provided for these containers against logical attacks on various cards. Most cards we analysed do not seem to provide any such measures. On the cards which do protect keys and PINs, notably by encrypting them, we present new attacks which break this protection.

Section 2 presents related work on logical attacks on Java Card. Section 3 presents our security analysis of key and PIN containers against logical attacks using malicious code. It also presents a new attack on the global APDU buffer exploiting type confusion, and the results of the study of undocumented opcodes. Section 4 summarizes the evaluation of five Java Cards from different manufacturers. More details about the attacks can be found in the first author's Master thesis [17]. Finally, Section 5 discusses countermeasures presumably implemented on some cards and proposes improvements.

2 Background and related work

In logical attacks, malicious inputs are sent to a Java Card applet to exploit flaws in the implementation of the applet or the underlying platform. Logical attacks are also possible with malicious code rather than malicious input: installing malicious, ill-typed code can break the security guarantees of the Java Card platform provided, notably type- and memory-safety: a malicious (or buggy) applet can then compromise the integrity or confidentiality of data belonging to other applets or the platform itself, or even compromise the integrity of their code.

Since the first paper to discuss logical attacks on the Java Card platform [18], and the first systematic overview of logical attacks and countermeasures that various cards implement [11], there is now a considerable amount of literature on logical attacks on Java Cards, e.g. [7, 9, 1, 6]. We do not discuss all this literature here, but focus on the related work that presents tricks that we also used in our attacks.

The simplest logical attacks just rely on ill-typed code and exploit the resulting type confusion to get illegal access to memory. The bytecode verifier is optional (except on the Java Card 3 Connected editions), and on cards without a bytecode verifier ill-typed code can in principle be installed. However, this does make very strong assumptions about the attacker and the card: many cards do include a bytecode verifier, and even if a card does not, installing applets will be tightly controlled with digital signatures using Global Platform. If ill-typed code can be installed, it can break the platform security guarantees of type- and memory-safety. Note that also the runtime measures of the firewall can be completely bypassed by ill-typed code. Additional runtime checks by a defen-

sive platform implementation can mitigate this, and some cards do take such measures, as experiments with cards confirm [11].

A more subtle variant of ill-typed code is to use incompatible shareable interfaces [18]; this involves a pair of applets, each of which is well-typed in isolation. Another way to create type confusion is to exploit flaws in platform implementation, e.g. flaws in the implementation of the transaction mechanism [11], or flaws in the on-card bytecode verifier [10].

Once illegal memory access can be achieved, modifying meta-data of objects is a standard trick to escalate the impact of the attack. The classic example is to change the length field of an array, which is recorded as metadata in the representation of array objects, to get access to more data than allowed. Other tricks include pointer arithmetic to corrupt or spoof references and the use of the `getstatic_b` instruction. This last instruction allows malicious code to access arbitrary memory locations [8], as it bypasses the runtime checks of Java Card firewall. The ultimate goal of a memory attack is to obtain a full memory dump [7, 4].

A more advanced way to escalate the impact of illegal memory access is to modify the code of other applets. Iguchi-Cartigny and Lanet present a Trojan applet which scans the memory and replaces instructions in the code of other applets to modify their behaviour [9]. To bypass bytecode verification, the authors use a byte array which contains the correct byte-level representation of a method body for a method with malicious code. As this byte array is data, not code, its content is not inspected by the bytecode verifier. However, by obtaining the address of the array and modifying the CAP file, the linker can be made to resolve method invocations to execute the data as a code.

Bouffard and Lanet present a way to execute 'shellcode' on the card and access the ROM memory of the card [4]. They also present a tool to identify native and Java Card bytecode in a memory dump.

Apart from attacking heap memory, attacks can also try to corrupt the stack. Faugeron presented a stack underflow attack on a Java Card VM [6]. The attack uses the instruction `dup_x`, which copies up to four words from the top of the stack. If the implementation of this instruction does not check for stack underflow, which is the case on some cards, the instruction can read outside the current stack frame, which allows access to data belonging to other methods located on the stack.

The only paper in the literature that looks at secure containers is by Farhadi and Lanet [5]. Here an attack is presented to access the plaintext value of a 3DES key belonging to another applet. The authors use a novel method to exploit type confusion, using the API method `arrayCopyNonAtomic`. The authors propose a number of countermeasures in the paper. In particular, they suggest encrypting the key container using the secret key which is not stored in non-volatile memory. The authors suggest that some cards do in fact use this protection. This is indeed the case, as we confirm in this paper, but we also show that it may be possible to bypass such protection.

3 Logical attacks on secure containers

This section presents new logical attacks on secure containers that the Java Card platform provides for cryptographic keys and for PIN objects. The goal of these attacks is for one (malicious) applet to get illegal access to the content of such secure containers of another applet, with an ultimate aim to obtain the plaintext keys and PINs, or to reset the PIN try counter, which enables brute-force guessing of PINs.

These attacks assume that the attacker is able to install his own applet on a card. Using the techniques discussed in Section 2 the attacker can then try to illegally access memory, but here the attacker can be stopped by runtime countermeasures. For the secure containers the attacker may run into countermeasures designed to protect the confidentiality of keys and PINs, the integrity of the PIN try counter, and possibly also the integrity of keys and PINs.

The attacks described below require an attacker to get knowledge about secure container implementations and their representation in memory. For this, an attacker can install an applet which uses the containers he wants to study on a card and then use techniques as described in Section 2 to access the raw memory, which will reveal the internal memory representation of these containers. The attacker can then reverse-engineer the memory representation of the containers and try to locate such objects in the memory belonging to other applets.

Of course, if the platform does not provide any countermeasures to protect the confidentiality of PINs and keys – by storing the plaintext – or to protect the integrity of try counters, then it is clear that such attacks are possible. Indeed, it turns out that many cards store PINs and keys in plaintext format, unencrypted, and try counters without any redundancy or integrity checks.

The attacks presented in sections 3.1-3.3 below bypass defensive measures we found implemented in cards, namely simple integrity checks of try counters and encryption of keys and PIN with an unknown platform key.

Section 3.4 introduces the attack on APDU buffer which allows an attacker to store and reuse a reference to a global array, and Section 3.5 discusses the investigation of unspecified opcodes that some VMs turn out to support. An overview of all attacks on the different cards is given in Section 4.

3.1 Changing PIN try counters

The API class `OwnerPIN` offers applet developers a standard implementation for PIN objects, which store PIN codes and then provide all the associated functionality, including the administration of the number of tries left for guessing the PIN in a so-called try counter.

Providing this implementation of PIN codes in the platform is not only convenient, it also avoids the risk of insecure ad-hoc implementations. Moreover, the API implementation can take into account platform level weaknesses or countermeasures, also regarding physical attacks. Indeed, the API specs say that the implementation of the `OwnerPIN` class should be secure against some attacks, including attacks that try to abuse the transaction mechanism.

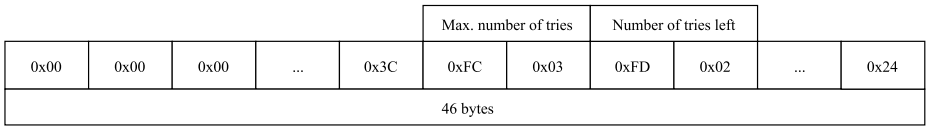

Fig. 1. Memory representation of an `OwnerPIN` object on `card_a`

Of the five cards we evaluated, only on one card, `card_a`, could we confirm that it implemented integrity checks of the `OwnerPin` try counter. However, as explained below, this integrity check could be reverse-engineered and bypassed.

Implementation of the attack Our attack used a malicious applet to access an arbitrary memory location by manipulating metadata of array objects, as described in [11]. We could then observe the raw memory representation of an `OwnerPIN` object, which is presented in Figure 1.

It is clear that the data structure does not contain the PIN in a plaintext. Since the Java Card platform does not specify this, it is not easy to say where the PIN is actually stored. However, it is easy to see four bytes in the data structure that record the maximum number of tries and the number of tries left.

The card uses a redundant representation for the these counters, using two bytes for each of these counters to implement an integrity check. The second byte stores the counter value itself and the first byte stores its bit-level compliment, so that XORing the two bytes always yields `0xFF`.

This technique is widely used to prevent fault injections. Faults that change a memory cell to a random value are clearly dangerous for these counter values used in PIN objects, as setting them to random values is likely to result in more tries than intended. Indeed, we found that if we changed any of these bytes and broke this integrity check, then the card stops responding permanently.

These integrity checks are very effective against fault attacks, but not against the logical attacks we consider: once the attacker knows how the checks work, it is easy to use a logical attack to corrupt the counter values. Indeed, by adapting our malicious applet to reset the try counter to maximum values, we could easily brute force the PIN, by repeatedly guessing the PIN and then invoking our malicious applet to reset the try counter. Creating an applet with an `OwnerPIN` object and an interface allowing to input a PIN a four digit PIN could be brute forced within 15 minutes.

3.2 Retrieving plaintext DES keys

The Java Card API also provides classes to store and use cryptographic keys, e.g. the class `DESKey` for DES keys. Methods of this class include `setKey(byte[] keyData, short kOff)` and `getKey(byte[] keyData, short kOff)` for setting and getting value of the key. When setting a key, it is possible to supply the

key in encrypted form[3]. Note that *supplying* a key in encrypted form does not say anything about whether keys are *stored* in an encrypted form: whether an API implementation chooses to store keys in an encrypted form is not specified, and indeed the user of the API cannot tell.

The method `getKey` always returns the plaintext key. From a security perspective, supporting this operation at all can be criticised [16], and indeed, we use it in our attack.

Using a malicious applet, one can inspect raw memory to see if keys are stored in plaintext format. Of the cards, we evaluated, only one card, `card_a`, stored keys in encrypted form. However, we could craft an attack which allows a malicious applet to retrieve the plaintext DES key of another applet, as described below.

Implementation of the attack To perform the attack, the malicious applet reads the raw byte representation of an encrypted DES key from the memory of another applet and copies it to a DES key object of its own. Once the encrypted key is copied, the attacker can then simply call the `DESKey.getKey()` method to get the plaintext. Figure 2 sketches this attack. So apparently on this card the same card encryption key is used to encrypt all DES keys of all the applets.

Note that we have to copy the content of the victim applet's key object to one owned by the malicious applet because the firewall prevents the malicious applet from invoking the `getKey()` method on a Key object belonging to another applet.

A – relatively simple – countermeasure against this attack would be to diversify the keys used for key encryption for different applets. A simple and cheap diversification operation, say XORing the master key with the AID of the applet, would already make the attack unfeasible. We did confirm that different cards of the same type use different encryption keys, so at least the platform implementation does not hardcode the same key on all cards.

As noted in [5], ideally the encryption key would be stored in memory that would be hard or impossible for an attacker to access (e.g. in ROM or memory of the crypto co-processor). We did not reverse engineer the card to the point that we could tell that this key is indeed not stored anywhere in non-volatile memory, but using the attack above we do not need to know this key anyway.

3.3 Retrieving plaintext PINs

As described in Section 3.1, with illegal memory access we could modify the try counter and then find out a PIN by brute force guessing, even though the PIN was apparently stored encrypted. However, this attack is quite slow.

The class `OwnerPIN` does not provide a method `getPIN()`, similar to the `getKey()` of `DESKey` class, to retrieve plaintext PINs. From the point of view of

[3] The factory method `KeyBuilder.buildKey` for creating `Key` objects provides support for this, for those `Key` subclasses that implement the `javacardx.crypto.KeyEncryption` interface.

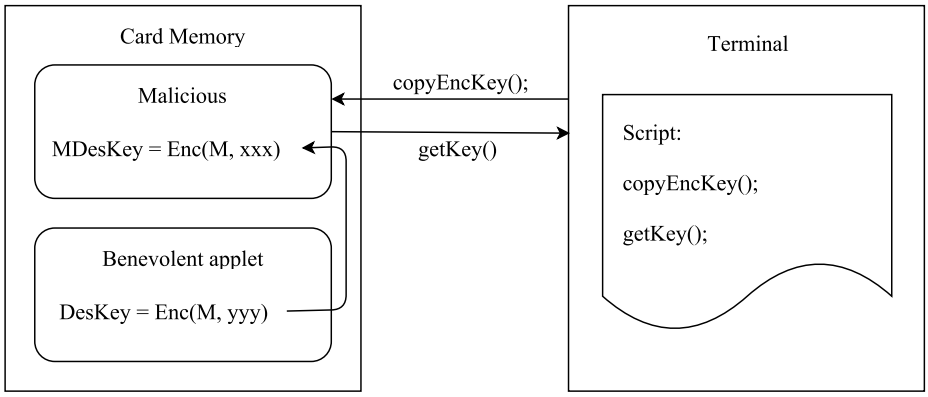

Fig. 2. `DESKey` decryption attack: from the terminal we invoke two operations on the malicious applet, the first to copy the encrypted key to a `MDesKey` object owned by the malicious applet, and a second to extract the plaintext key.

security, providing such a method is clearly a bad idea, and from the point of functionality there is no sensible reason for having it. If such a `getPIN()` method was available, we might undo the encryption of PINs in the same way that we undid the encryption of DES keys in Section 3.2, by copying the raw encrypted byte representation of the PIN to another PIN object and invoking `getPIN()`.

Still, as the content of both PIN and key objects stored encrypted on the card, one possibility is that the same algorithm is used in both cases. Indeed, it is easy to notice that both 8 byte long DES keys and 8 byte long PIN are stored as 10 bytes, which suggests this may indeed be the case.

This suggested an attack where the `getKey()` operation of the `DESKey` class is used to retrieve plaintext PINs: a malicious applet copies the encrypted representation of a PIN belonging to another applet to one of his own DES key objects, and then invokes `getKey()` to retrieve the plaintext. Figure 3 sketches this attack.

We found this attack worked. So apparently the same encryption key is used not only for all keys but also for all PINs, for all applets. This provides an attacker with the ability to encrypt and decrypt any values using this key, without actually knowing it. Again, some key diversification would thwart this attack: just using a different key for encrypting PINs than is used for encrypting keys would suffice.

3.4 Illegal access to APDU buffer array reference

As part of the firewall functionality, the Java Card runtime environment restricts storing references to so-called global arrays, because they are potentially shared amongst applets. The APDU buffer is such a global array, and the runtime environment specification states:

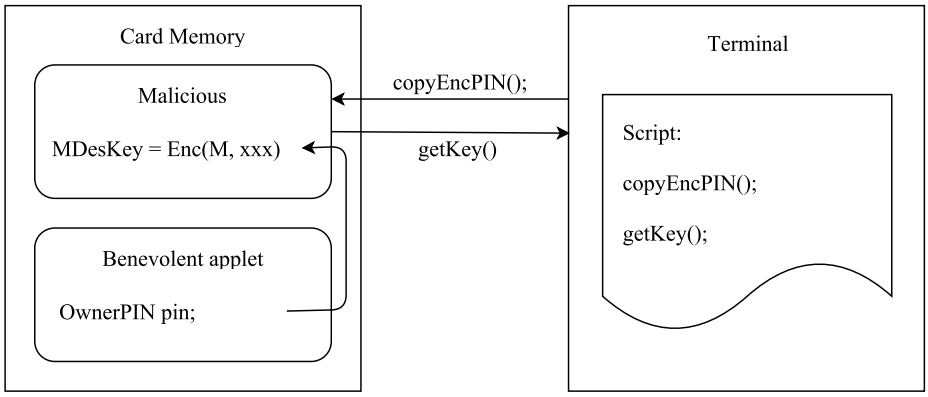

Fig. 3. `OwnerPIN` decryption attack

> "All global arrays are temporary global array objects. These objects are owned by the JCRE context, but can be accessed from any context. However, references to these objects cannot be stored in class variables, instance variables or array components. The JCRE detects and restricts attempts to store references to these objects as part of the firewall functionality to prevent unauthorised re-use." [14, §6.2.2]

This means that references to global arrays cannot be stored in persistent memory on the heap (which is where class and instance variables are stored). Such references can only be stored in local variables, i.e. on the stack, which is in transient memory. (Moreover, to avoid data leakage between applets, the JCRE and API specs state that the APDU buffer must be cleared to zeroes whenever an applet is selected [14, §6.2.2].)

However, if an attacker can (illegally) cast a reference to a short value and back, then by storing that short value he can bypass the restriction on storing references to global arrays. We found that this was possible on all the cards we analysed. This effectively breaks the requirement quoted above.

The illegal access to the APDU buffer this allows and the attacks that this in turn enables are extensively discussed by Barbu et al. [2].

Implementation of the attack In order to bypass the restriction on storing references to global objects, a malformed applet was developed. The core functionality of the applet is given by the functions below:

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

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

```
    return (byte[]) addr;
}
```

The code in the listing above is clearly ill-typed, so it will be rejected by any compiler, but a CAP file for this code can be created by hand, and this CAP file can be installed on a card if it does not have an on-card bytecode verifier.

Once the code is installed, the method `addr()` allows a reference to any array, including a global array, to be turned into a short, and the platform will not prevent this short from being stored in a class or instance variable. At any point in time later, the method `ptr()` can be used to turn this stored short back into a reference, so this is functionally equivalent to storing the reference to the global array in a class or instance variable. This is illustrated in more detail in the code samples below.

First, the code below illustrates how the firewall will disallow storing a reference to a global array, the APDU buffer, in a class variable `buffClassCopy`, but will allow storing it in a local variable `buffLocalCopy`:

```
public class App extends Applet {
    byte[] instance;
    ...
    public void process(APDU apdu) {
        byte[] local;
        byte[] buffer = apdu.getBuffer();
        local = buffer; // allowed by the firewall
        instance = buffer; // forbidden by the firewall
    }
}
```

However, using the operations `addr()` and `ptr()` we can obtain equivalent functionality without breaking the firewall rules:

```
public class App extends Applet {
    short instanceShort;
    ...
    public void process(APDU apdu) {
        byte[] buffer = apdu.getBuffer();
        instanceShort = addr(buffer); // allowed by firewall
    }
}
```

In any method where we would want to use `instanceShort` as a reference to a byte array, we can declare a local byte array variable and use `ptr()` to convert the short back to a reference: C

```
byte[] localVar;
localVar = ptr(instanceShort); // allowed
...// localVar can now be used, and points to the global array
```

It is worth mentioning that we observed some non-trivial countermeasures on `card_a` to prevent references to global arrays being stored in class or instance variables. If we attempt to store `ptr(instanceShort)` in an instance variable of type `byte[]`, the card terminates the session. There appears to be a runtime check so that when any class or instance variable is set to point to the APDU buffer, the card terminates the session.

Obviously, such a check uses resources and it is completely useless against the attack presented here: storing the reference as a short in a class variable is just as good as storing it as a reference.

3.5 Execution of 'illegal' opcodes

According to the Java Card virtual machine specification, [13] correct Java bytecode must include only instruction bytes from `0x00` to `0xB8`, `0xFE` and `0xFF`. The first range of opcodes is reserved for the standard bytecode operations. The last two opcodes are reserved for internal use by a virtual machine implementation, for any implementation-specific functionality.

Execution of illegal code on the cards revealed that one of the studied cards, namely `card_a`, supported opcodes which are outside of the range above, which we will call 'illegal' opcodes, for unspecified operations of the Java Card virtual machine.

Like the known and specified opcodes, illegal opcodes may be followed by zero or more byte parameters in the code (i.e. in the CAP file), and they can pop parameters from the top of the operand stack. In the case of the legal opcodes, the number of the code and stack parameters is defined in the specification. Finding out the number of parameters required by each of the illegal opcodes is the first challenge on the way to understanding the purpose of the illegal opcodes. We tackle this by executing an illegal opcode followed by a number of instructions pushing a constant on the operand stack and analyzing the state of the stack after execution. It is possible to analyse the stack state after execution of an illegal opcode by popping remaining words from the stack and saving to local variables. Unfortunately, just knowing the number of the parameters an illegal opcode uses does not provide a lot of clues for understanding its purpose.

Luckily, there is an emulator provided by the manufacturer of the card which also supports execution of the illegal opcodes. Comparison of the execution traces of the emulator executing illegal and legal bytecodes provided an understanding of the purpose of the opcodes since some of the illegal opcodes result in exactly the same or very similar execution traces as legal ones. Table 1 presents legal code corresponding to some of the illegal instructions. It suggests that these additional opcodes are just convenient shorthand, presumably for more compact bytecode. The idea to use the unused opcodes as abbreviations for common patterns is already discussed in [3].

A study of the illegal opcodes seemed to be promising, but it turned out to be very time-consuming and did not result in a successful attack. Still, a single bytecode instruction can be very dangerous, as illustrated by the power that `getstatic_b` gives to malicious code, so any additional opcodes deserve close

Illegal opcode	Equivalent instruction
0xBD	sload 4
0xBE	sload 5
0xBF	sload 6
0xC0	sload 7

Table 1. Corresponding legal instructions to illegal opcodes

Card	Global Platform	Java Card
card_a	GP 2.1.1	JC 2.1.2
card_b	GP 2.1.1	JC 2.2.1
card_c	GP 2.2.1	JC 3.0.4
card_d	GP 2.1.1	JC 2.2.1
card_e	GP 2.1.1	JC 3.0.1

Table 2. Specification of the cards

attention in any security evaluation. The use of the emulator turned out to be a great tool in hands of an attacker to help understand the internal design of the Java Card implementation.

4 Evaluation

We tried out our attacks on five different cards from different manufacturers. Table 2 lists the version numbers of Java Card and Global Platform the cards provide. The outcome of the attacks is summarised in Table 3. Here ✓ indicates that an attack was successful, and ✗ that the attack failed. So the fewer ✗s a card has, the better its security is.

The techniques we used in our attacks, apart from those discussed in Section 3, were known techniques discussed in Section 2, namely

- modifying metadata (esp. of arrays),
- 'spoofing' references (by doing some pointer arithmetic),
- using `getstatic_b`,
- using 'illegal' opcodes, in a variation of the attack of [9], and
- accessing arbitrary objects as if they were arrays (effectively a generalisation of the attack with `arrayCopyNonAtomic` used in [5][4]).

For `card_a`, the only card on which we found countermeasures to protect secure containers, we used the techniques described in Section 3 to bypass these.

On `card_b` and `card_e` no countermeasures were implemented and all the data and metadata were stored in a plain text. As a result, the keys and PINs were easily recovered without using the more complex attacks described in Section 3.

[4] We found that the attack using `arrayCopyNonAtomic` did not work on any of the cards we had, so this seems an implementation-specific weakness of the particular card studied in [5].

Attack \ Card	card_a	card_b	card_c	card_d	card_e
Changing a PIN trycounter	✓	✓	✗	✗	✓
Retrieving a plaintext PIN	✓	✓	✗	✗	✓
Retrieval a plaintext DES key	✓	✓	✗	✗	✓
APDU buffer array reference	✓	✗	✓	✓	✓

Table 3. The results of the attacks on the various cards.

Most attacks failed on `card_c` and `card_d` because it was not possible to get unauthorised access to the memory of another applet. Although it was possible to install and run malicious code, type confusion was not possible, which might be a result of comprehensive defensive runtime type checks that these cards perform.

The APDU buffer attack failed on card `card_b` because this card refuses to install CAP files that contain library packages, which we use in this attack. Clearly, the APDU buffer reference attack worked on the largest number of cards. This underlines the observation in [2], that an applet developer should be aware of the risk that the APDU buffer may be compromised by other applets.

5 Countermeasures

The most obvious countermeasure against logical attacks with malicious code is of course to have an on-card bytecode verifier. Note that even on a card with an on-card verifier, bugs in the platform may still allow an attacker to bypass the protection it offers (as shown in [11, 10]).

In addition to a bytecode verifier, if the card's hardware provides a memory management unit to control access to memory areas, that can be used as additional line of defence against logical attacks.

Runtime checks can make logical attacks harder. Doing full type checking at runtime would stop any ill-typed code, but this is prohibitively expensive. In practice, cards do some checks at runtime. On two of the cards we studied, `card_c` and `card_d`, all our attempts to access secure containers with malicious code failed due to defensive runtime checks by the platform. When experimenting with attacks cards will sometimes mute because integrity checks are violated. In all, 24 cards permanently muted in the course of our experiments. Of course, which integrity checks are performed is unknown, and can only be figured out by trial and error: The Java Card specifications do not prescribe any such countermeasures, and this is completely up to the implementation. Many of the papers mentioned in Section 2 speculate about countermeasures that were encountered.

One implementation choice in a platform that can have a big impact on the capabilities of ill-formed code is how references are represented. Some implementations simply use pointers (i.e. memory addresses) for references. However, most modern cards use indirect referencing using an indirection table. Here a reference is an index in some table that stores the real pointer value (i.e. the

memory address where the data is located). Using an indirection table makes attacks that corrupt or spoof references harder: it prevents an attacker from performing arithmetic operations on references in order to get access to the random location of the memory. Some of the manufacturers that implement indirect referencing store the metadata of an array next to data, instead of storing it in the reference table. Since array metadata has a fixed size it does not seem to be too difficult to store it in the table. Storing metadata in the reference table would make it much harder for an attacker to corrupt this data. For example, modifying memory outside the array bounds would not hit metadata of adjacent array objects in memory.

According to Java Card virtual machine specification memory addresses of a virtual machine are 16 bits. This means only 64 KB of non-volatile memory can be addressed by an applet using Java Card instructions. Nowadays many Java Cards have bigger non-volatile memory than 64 KB. This means there is unaddressable memory, that cannot be accessed by a malicious applet unless it can execute native code. Although it has been shown to be possible to execute native code on some Java Cards [4], this is not an easy task. Clearly, storing the reference table with object metadata in such memory that is not addressable by (non-native) code running on the virtual machine would be a good countermeasure to make logical attacks harder.

The reason we could retrieve plaintexts of data in the secured containers for `card_a` was that the same encryption key was used for all containers, i.e. both keys and PINs, and for all applets. This vulnerability could be avoided by using key diversification in such a way that for each applet, or even each object, a different key is used. There are many options for such a diversification scheme. For example, the AID of an applet could be used diversify a master encryption key to obtain a key for protecting objects of that applet. Of course, relying on the AID for the key diversification assumes that the attacker cannot manipulate the AID of his attack applet. An even stronger solution would be to use the memory address of the object as a parameter to diversify a key to obtain a unique key to protect a specific object. However, this measure would complicate re-allocation of objects; it would therefore not be a good choice if you want the platform to support memory compaction as part of garbage collection.

Protecting the integrity of try counters from logical attacks seems harder than protecting the confidentiality of keys or PIN. Whatever integrity measure is implemented (say a cryptographic integrity check with a keyed MAC using the same key used to encrypt the PIN), an attacker with access to memory may simply be able to restore an earlier value of a try counter together with all the right integrity measures. The integrity check could of course be made a lot harder to reverse-engineer, and spreading out the data involved to represent these counters and any associated integrity checks in memory would make life much harder for an attacker.

The reason that the global array reference attack is possible on most cards is that the operand stack on the cards is untyped and it is quite difficult to prevent the attack without runtime type checks. A possible solution could be to

use a typed operand stack (but that comes with a big performance overhead) or separate stacks for references and other data types.

As noted by Barbu et al. [2], who extensively discuss attacks involving illegal access to the APDU buffer, here the applet developer should take into account that the APDU buffer can be compromised by other applets. It would seem to us that the risk this introduces (if any, as it assumes the presence of malicious applets) is greater on the new JavaCard 3 Connected Edition, because that supports multi-threading, which increases the possibilities for another applet to simultaneously access the shared APDU buffer.

6 Conclusions

Apart from explicitly specified security mechanisms such as the applet firewall and transaction mechanism, there are a number of additional countermeasures that can be implemented on Java Card smartcards. But, as it was shown in this paper, not all the countermeasures are equally effective. This underlines the importance of a bytecode verifier and the control on applet installation using digital signatures with Global Platform as lines of defence.

Our study showed that most cards we analyzed do not implement any special protection mechanism to protect secure containers from logical attacks with malicious, as they simply store keys and PINs in a plaintext. Even on cards which encrypts secret keys, we found that attackers can easily decrypt the ciphertexts by (ab)using the decryption functionality offered by the API itself through the `getKey()` method. The cards we had access to for our evaluation are engineering samples, so we cannot exclude the possibility that cards used in the field have different countermeasures.

It is worth mentioning that the encryption of the keys and PINs and the simple integrity check on try counters that we encountered may be aimed at preventing side-channel attacks rather than logical attacks. Against fault injections, these mechanisms would be very effective, both against fault attacks where the attacker tries to corrupt the try counter and against fault attacks where the attacker tries to modify PINs or keys to predictable values. (The latter requires *stuck-at* fault attacks where the attacker can predict that the value of a memory location after the attack will be a fixed value, e.g. all zeroes or all ones.)

As discussed in Section 5, a simple tweak of the scheme for encrypting keys and PINs – namely, adding some key diversification – would make the protection much stronger. Securing PIN try counters from logical attacks seems the harder problem.

References

1. Barbu, G., Duc, G., Hoogvorst, P.: Java Card operand stack: fault attacks, combined attacks and countermeasures. In: Smart Card Research and Advanced Applications (CARDIS), pp. 297–313. Springer (2011)

2. Barbu, G., Giraud, C., Guerin, V.: Embedded eavesdropping on Java Card. In: Information Security and Privacy Research, pp. 37–48. Springer (2012)
3. Bizzotto, G., Grimaud, G.: Practical Java Card bytecode compression. In: Proceedings of RENPAR14/ASF/SYMPA (2002)
4. Bouffard, G., Lanet, J.L.: Reversing the operating system of a Java based smart card. Journal of Computer Virology and Hacking Techniques 10(4), 239–253 (2014)
5. Farhadi, M., Lanet, J.L.: Chronicle of a Java Card death. Journal of Computer Virology and Hacking Techniques pp. 1–15 (2016)
6. Faugeron, E.: Manipulating the frame information with an underflow attack. In: Smart Card Research and Advanced Applications (CARDIS), pp. 140–151. Springer (2013)
7. Hogenboom, J., Mostowski, W.: Full memory read attack on a Java Card. In: 4th Benelux Workshop on Information and System Security Proceedings (WISSEC09) (2009)
8. Hyppönen, K.: Use of Cryptographic Codes for Bytecode Verification in Smartcard Environment. Master's thesis, University of Kuopio (2003)
9. Iguchi-Cartigny, J., Lanet, J.L.: Developing a Trojan applet in a smart card. Journal in Computer Virology 6(4), 343–351 (2010)
10. Lancia, J., Bouffard, G.: Java card virtual machine compromising from a bytecode verified applet. In: Smart Card Research and Advanced Applications (CARDIS). pp. 75–88. Springer (2015)
11. Mostowski, W., Poll, E.: Malicious code on Java Card smartcards: Attacks and countermeasures. In: Smart Card Research and Advanced Applications (CARDIS), pp. 1–16. Springer (2008)
12. 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. ACM (2005)
13. Sun Microsystems: Java Card 2.1.1 Virtual Machine Specification (2000)
14. Sun Microsystems: Java Card 2.2.2 Runtime Environment Specification (2006)
15. Vermoen, D., Witteman, M., Gaydadjiev, G.N.: Reverse engineering Java Card applets using power analysis. In: Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems, pp. 138–149. Springer (2007)
16. Vétillard, E.: Should we deprecate DESKey.getKey()? (2007), http://javacard.vetilles.com/2007/06/19/should-we-deprecate-deskeygetkey/, blog entry
17. Volokitin, S.: Good, Bad and Ugly Design of Java Card Security. Master's thesis, Radboud University Nijmegen (2016)
18. Witteman, M.: Java Card security. Information Security Bulletin 8, 291–298 (2003)