Attacks against Smart Cards: Hands On Session

Jean-Louis Lanet
SSD - XLIM Labs, University of Limoges,
87000 Limoges, France
Email: jean-louis.lanet@unilim.fr

***Abstract*—Smart card are often the target of software or hardware attacks. Recently several logical attacks have been developed that allows to dump the EEPROM memory. This kind of attack are particularly affordable for students who can learn reverse engineering techniques on devices known to be tamper resistant. This tutorial will demonstrate how with a few material a graduate student within a couple of hours is able to reverse an application.**

***Index Terms*—smart card, logical attack, shell code, counter measures, reverse engineering**

I. INTRODUCTION

Java Card is a kind of smart card that implements one of the two editions, "*Classic Edition*" or "*Connected Edition*", of the standard Java Card 3.0 [1]. Such a smart card embeds a virtual machine which interprets codes already romized with the operating system or downloaded after issuance. Due to security reasons, the ability to download code into the card is controlled by a protocol defined by Global Platform [2]. This protocol ensures that the owner of the code has the necessary authorization to perform the action. Java Card is an open platform for smart cards, *i.e.* able of loading and executing new applications after issuance. Thus, different applications from different providers run in the same smart card. Thanks to type verification, byte codes delivered by the Java compiler and the converter (in charge of giving a compact representation of class files) are safe, *i.e.* the loaded application is not hostile to other applications in the Java Card. Furthermore, the Java Card firewall checks permissions between applications in the card, enforcing isolation between them.

Java Card is quite similar to any other Java edition. It only differs (at least for the *Classic Edition*) from standard Java in three aspects: i) restrictions of the language, ii) run time environment and iii) applet life cycle. Due to resource constraints the virtual machine in the *Classic Edition* must be split into two parts: the byte code verifier executed off-card is invoked by a converter while the interpreter, the API and the Java Card Run time Environment (JCRE) are executed on board. The byte code verifier is the offensive security process of the Java Card. It performs the static code verifications required by the virtual machine specification. The verifier guarantees the validity of the code being loaded in the card. The byte code converter transforms the Java class files, which have been verified and validated, into a format that is more suitable for smart cards, the CAP file format. Then, an on-card loader installs the classes into the card memory. The conversion and the loading steps are not executed consecutively (a lot of time can separate them). Thus, it may be possible to corrupt the CAP file, intentionally or not, during the transfer. In order to avoid this, the Global Platform Security Domain checks the file integrity and authenticates the package before its registration in the card.

Some attacks have been successful in retrieving secret data from the card. Thus we will present different approaches to get access to data, which should bypass Java security components. The aim of an attacker is to generate malicious applications which can bypass firewall restrictions and modify other applications, even if they do not belong to the same security package. Several papers were published and they differ essentially on the hypotheses of the platform vulnerabilities and this tutorial will apply some of the vulnerabilities that can be exploited with an ill-typed applet.

II. *JAVA CARD SECURITY FEATURES*

To enforce protection between applets, classical Java technology uses the type verification, class loader and security managers to create private namespaces for applets. In a smart card, complying with the traditional enforcement process is not possible. On the one hand, the type verification is executed outside the card due to memory constraints. On the other hand, the class loader and security managers are replaced by the Java Card firewall.

A. Byte Code Verifier

Allowing code to be loaded into the card after post-issuance raises the same issues as the web applets. An applet not built by a compiler (hand-made byte code) or modified after the compilation step may break the Java sandbox model. Thus, the client must check that the Java-language typing rules are preserved at the byte code level. The Java is a strongly typed language where each variable and expression has a type determined at compile-time, so that if a type mismatches from the source code, an error is thrown. The Java byte code is also strongly typed. Moreover, local and stack variables of the virtual machine have fixed types even in the scope of a method execution. None of type mismatches are detected at run time, and that allows making malicious applets exploiting this issue.

Due to resource constraints, the Byte Code Verifier is never embedded into the card. It is often replaced by some static

checks during the load and some dynamic check during the execution. In order to built an efficient attack, we need to discover which checks are implemented and during which phase their are executed.

B. Firewall

The separation of different applets is enforced by the firewall which is based on the package structure of Java Card and the notion of contexts. When an applet is created, the Java Card Runtime Environment (JCRE) uses a unique Applet IDentifer (AID) to link it with the package where it's been defined. If two applets are an instance of classes of the same Java Card package, they are considered in the same context. There is a super user context, called the JCRE context. Applets associated with this context can access to objects from any other context on the card. Each object is assigned to an unique owner context which is the context of the applet created. An object method is executed in the object owner context. This context provides information allowing, or not, the access to another object. The firewall prevents a method executing in one context from access to any attribute or method of objects to another context.

C. The CAP File

As described by S. Hamadouche in [3], the CAP (for *Convert APplet*) file format is based on the notion of components. It is specified by Oracle [1] as consisting of following standard components:

1) The `Header`: contains main information about the CAP file and the defined package;
2) The `Directory`: list the size of each component;
3) The `Applet` (*optional*): if the CAP file contains at least one applet, this component has a reference to each defined applet else this component is empty;
4) The `Import`: describes the imported packages by the classes defined in the package;
5) The `Constant Pool`: contains each information about: classes, methods and fields referred by each element of the `Method` component of the CAP file;
6) The `Class`: describes each class and interface defined in the package;
7) The `Method`: contains each declared method in the package, the abstract methods defined by the classes and the exceptions' handler associated to each method;
8) The `Static Field`: has each information to create and initialize each static field defined in the package;
9) The `Reference Location`: contains an offset list, in the `Method` component to each element which has reference in the `Constant Pool` component;
10) The `Export` (*optional*): lists each static element which may be imported by the classes contain in other packages;
11) The `Descriptor`: gives needed information to analyze and check the CAP file components.

Moreover, the targeted Java Card Virtual Machine (JCVM) may support user `custom` components. We except the `Debug` component because it is only used on the debugging step and it is not sent to the card.

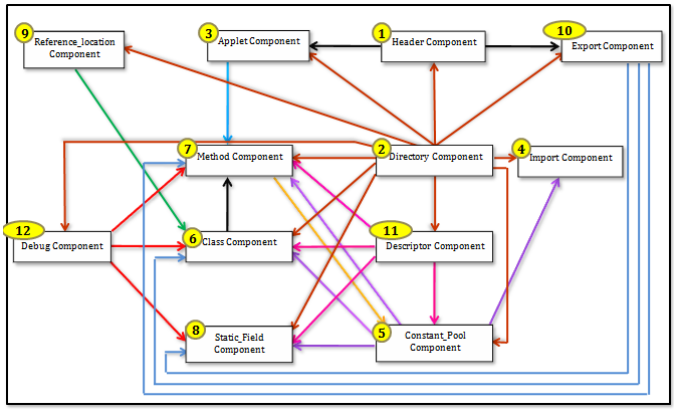

Figure 1: Interdependency links between each of CAP file component

As described in the figure 1, each component has a dedicated role and is linked to each others. A hand-modification of a component is difficult and may provide meaningless file. An invalid file is often detected during the installation step by the targeted JCVM.

III. LOGICAL ATTACKS

Logical attacks against smart card can be classified in two categories: ill-typed applications or well-typed applications. But the second category can also be split into permanent well-typed applications or transient well-typed applications. In ill-typed applications [4], [5], the input file has been modified in order to illegally obtain information. Permanent well typed application [6], relies on some weakness of the specification. Transient well-typed applications is a new research field [7], [8] were an application mutes when a fault occurs. In this direction, we have *fault enabled viruses*. Ill-typed applications and transient well-typed applications need to apply byte code transformation engineering at the CAP file level.

A. The EMAN2 attack

The attack consists in changing the index of a local variable. The specification says that the number of variables that can be used in a method is 255. It includes local variables, method parameters, and in case of an instance method invocation, a reference to the object on which the instance method is being invoked. For that purpose we use two instructions: `sload` and `sstore`. As described in the JCVM Specification , these instructions are normally used in order to load a short from a local variable and to store a short in a local variable.

So, if we change the operand of `sload`, says `sload 4`, which store a short into the Local variable 4. Imagine that your program store at the Java level a short value that corresponds to the first element of an array into the last local variable plus an offset of 2. It means that we try to store into a local that do not exist. Due to the fact that the Byte Code Verifier checks the range of the locals this must be detect during the conversion

process. But after this verification if one changes the value of the operand it will not be detect during run time.

If the short value represent the address of the first element of an Array, then this manipulation will change the return address of the current method. When existing from this method, instead of returning to the caller, the program will execute the byte array. Then the content of the byte array must be interpretable by the virtual machine without any stack under or overflow. We are able to execute an arbitrary shell code, hand written at the byte code level.

B. Sketch of the Tutorial

During this tutorial you will learn how to replay the EMAN 2 attack [5], get the addresses of the most important method of the API, generate a simple return code attack, then develop a rich shell code, dump an installed applet and discover some (inefficient) counter measure embedded in the card.

a) Writing the attack applet: During this phase you will design your first Java Card application, compile it, convert it (from `class` to CAP), load it into the card thanks to the OPAL library and run it. You will send commands to the card and receive the response. This is the regular job of a Smart Card application designer. Learn to load and unload applications in the card.

b) Manipulating the Java Stack: You will transform your CAP file in order to be able to recover addresses of instances inside the card, lure the linker by retrieving the exact addresses of the API in memory, add and remove byte code in the binary CAP. These are the first steps of the logical attacks: the byte code engineering phase.

c) Writing shell code: You will generate your first attack by replaying the EMAN2 attack. The main idea is to change an operand in the byte array in order to store the value of an array in the return address location. At that time, the array will be fill by a return byte code. You will verify that this hook has no visible effect on the program. And then dynamically you will change the content of the array with your own rich shell code able to download the content of the card.

d) Discovering the content of the card: Then you will navigate into the card memory, find objects, classes, arrays, methods and so on. May be you will find already installed applet... but also the basic countermeasures against this attack. You will have to discover by your self how to bypass these countermeasures.

C. Tools used to set up the Attack

This kind of attack is often based on CAP file modification and upload of hostile applet. To exploit this vulnerability we need tools to automate the process: the CapMap and OPAL.

e) The Cap File Manipulator (CapMap): In this section, attacks are based on an ill-formed CAP file. The CAP file, likely explained in the previous section, has several dependent components. In order to have an easy way to made the required modifications, we developed a Java-library which provides the modifications and corrections of dependencies on the CAP file. This open-source library [9] will be used during this tutorial with the on-line version of the CapMap.

f) OPAL: OPAL[10] is a Java 6 library that implements Global Platform 2.x specification. It is able to upload and manage applet life cycle on Java Card. It is also able to manage different implementations of the specification *via* a pluggable interface.

These libraries provide an efficient way to automate attacks with the analysis of the card responses and generate appropriate requests.

Attendees should master the Java language, have a Java compiler on their laptop have a Java IDE (Eclipse,...). They should upload the OPAL library before attending the tutorial.

IV. EXPERIMENTS

Each attendee will use a reader and a smart card. Each smart card has a number. Cards and reader must be return to the instructor at the end of the tutorial session.

First of all, you need to set up the environment. You need a compiler chain to be installed with the OPAL library thta you can download at the following address https://bitbucket.org/ssd/opal/downloads/. Download the version **opal-library-xxxx-SNAPSHOT-jar-with-dependencies.jar** (xxxx being the latest commit version). Add this library in your classpath project of your IDE. The Java source files and the compilation scripts are available at the following address: *http://cartes.msi.unilim.fr* **Download Applets source code**, it contains a zip file applets_crisis.zip. Log into the web service using the user name crisis12_n where n is the number written on the smart card, the password is the same (example crisis12_4 for the card labeled 4).

In this file, you will find the applets Crisis1, Crisis2, Crisis3, a helloworld applet and four client applications to run on your laptop for communicating with the smart card. The Java program Crisis1 has to be used with the applet Crisis1 and so on. The program Simple_OPAL_Main.java has to be used with the hello world applet. In these client programs, you have to modify the line 95 installApplet (PACKAGE_ID, APPLET_ID, APPLET_ID, "TODO:YOUR_FILE"); the string by the name of your CAP file (the Java Card binary file format).

The gencapfile is the script (.bat) for windows platform or (.sh) for linux platform. Extract the zip and store it in your laptop. These scripts will be used to compile and convert the Java Card Applet. It will generate the CAP files in each applet directory.

You have access to the CapMap through a web service at the following address :http:// cartes.msi.unilim.fr. With this web service you can **Upload** a CAP file onto the server and read the binary file clicking on **Overview**. You can edit each method by clicking on **Methods**. Chose the method you want to edit on the top left using the lift then click on the **Ok** button. In the *Opcodes* area you can modify the opcode, and in the *Arguments* area you can modify the parameters of the opcodes and once you click either on **Submit** or **Enter** your modifications are taken into account. If you read your binary file with e bf Overview button you can see that your

modifications have been taken into account, you can download back your modified file using the **Download** button. Take care if you modify the opcodes or the arguments in a way the program is no more executable (undefined byte code,...) you will raise an exception.

A. Step 0: checking the compiling and disassembling chain

Compile and convert using the script with a double click on the bat file (either the .bat or .sh file), the simple applet **Hello world** to verify your tool chain. It generates CAP files in a new directory called Javacard for each applet. Once compiled and converted use the Simple_OPAL_Main.java program to upload it into the card, send the command apdu and verify the response. Normally it should send the "Hello world" byte array to the card which sends back the same byte array. Reaching that point, you have validated your Java Card tool chain and you are ready to try to hack the card.

Possible troubleshooting

- Wrong version of Java, the script is for the latest version of Java says 1.7.0_07 if you use an earlier version, modify the script file. To know your version use java -version.
- User of a 64 bits machine using a 32 bits Java: modify in the batch file the address of the jdk: C:/program Files (x86)/Java/jdk...
- User of Eclipse, it could not be able to find the API smartcardio: in Project properties, in Errors/Warnings click enable project specific settings and then in Deprecated and restricted API, change Forbidden reference from Error to Warning.

B. Step 1: retrieving the address of an array

You will have to write your shell in an array. Even if it is not mandatory, the first consists in basic stack manipulation using the CapMap web service.

- Edit the Crisis1 applet in your favorite Java editor. Find the method getMyAddress(). Compile it, verify it, convert it. In fact this step has already be done with the script during the previous step but if you modify the source code you have to redo it.
- Edit the resulting CAP file with the CapMap editor,
- Find the right method (one where you find the instruction sspush 0xCAFE), nullify the byte code in order to push on top of the stack the reference of the array, recover the CAP file,
- Load the application into the card, send the adequate APDU and obtain the address of the array.

C. Step 2: perform an EMAN2 attack to retrieve the address

At that point, you have obtained the linked addresses of the API methods, you need to transform a new applet in order to modify the return address of a called method. The return address will be the reference of your array.

- Edit the Crisis2 client in your favorite Java editor, it initialize (sendAPDU) your shell code with the minimum to avoid a crash: it raises the exception 0x6789, it obtains the address of the array, third send back the address to the card,
- Edit the Crisis2 applet in your favorite Java editor. Three commands corresponding to the three APDU : INS_SET_SHELLCODE which initializes the shell code (already done in the array declaration), INS_GET_SHELLCODE_ADDRESS: return the array address (Warning you have to redo the step 1), INS_EXECUTE_SHELL_CODE that patches the card. Look at the method executeOurShellCode. The short array_address is the one sent to the client and obtained at step 1. Compile it, verify it, convert it.
- Edit the resulting CAP file and modify the index of the local variable in order to store it at the return address location of the array which is the (max local + arg)+ 1 (in case of doubt ask the instructor, this could be a cause of the card suicide),
- Load the application into the card, execute the client on the laptop it sends the three APDU the last one launching your shell code. You are now able to store in the array any shell code you want.

D. Step 3: discovering the addresses of the API

We provide you only one API address: ThrowIt() : invokestatic 0x08c6. You will latter have to write linked shell code in the previously specified array. To obtain the addresses of the methods use the API discovering process.

- Edit the Crisis3 applet applet in your favorite Java editor. Look at the methods getTheISOExceptionThrowIt() and getTheAPDUSetOutgoingAndSendAddress(). Compile it, verify it, convert it.
- Edit the resulting CAP file and remove the tokens in order to push on top of the stack only the addresses that will be resolved by the linker,
- Run the client program Crisis3 which send the APDU to the card store the result. You have obtained the address of the two API methods. If you want other addresses for this card you have just to code the method.

E. Step 4: execute your rich shell code

Reaching that point you have hooked the card and you are able to execute any arbitrary array in the card. Dump the memory.

- In the array built a linked application that get a static value, store it in a buffer and increment the address of the static.
- When the buffer of 250 byte is full store it in the APDU buffer send the response, built the adequate structure to dump the memory from the address xxx to yyy,
- Run the external program xxxx which send the APDU to the card store the result.

F. Step 5: read the content of the array

Great you have dump the Java Card memory !!! Now pay attention to the structure.

- Use an hexadecimal editor and find the array stored at step one. What can you remark ?
- Find the countermeasure...

G. Step 6: find our own linked code in the memory

Now you know that addresses are scrambled when push on top of the stack. It doesn't matter the VM descrambled it for you for free. Nevertheless each time something is easy to do shall be protected in the card.Can you reverse a code ?

- Use an hexadecimal editor and find our own code.
- Look at the structure of the stored byte array far from what you sent to card ?

At that point you have learn how to dump the memory of the card, you saw that smart card manufacturers try to hide information but you can bypass it, you are able to find applet structures and modify other binary applications by removing, adding code. Keep in mind you have play with development card not a product. Do you believe a product is far from that ? What you can do with the shell code is just limited to your imagination. Enjoy !

V. CONCLUSIONS

This tutorial has presented the basics step to set up a logical attack against Java based smart card. This hands on session uses smart card sell on official web site. These cards are used for different markets such as e-Government, mobile communication, public transportation, pay TV... It has been demonstrated that several vulnerabilities remain into the virtual machine. Attendees have been able to reverse some part of the memory using an EMAN 2 attack. This attack is generic and run on most of the current cards sell on web sites or by major smart card manufacturers.

ACKNOWLEDGMENT

A special thank to Guillaume Bouffard who largely contributed to this hands on session and the students of the University of Limoges who verified every steps.

REFERENCES

[1] Oracle, "Java Card Platform Specification," http://java.sun.com/javacard/specs.html.
[2] Global Platform, "Card Specification v2.2," 2006.
[3] S. Hamadouche, "Étude de la sécurité d'un vérifieur de Byte Code et génération de tests de vulnérabilité," Master's thesis, Université de Boumerdés, 2012.
[4] J. Iguchi-Cartigny and J. Lanet, "Developing a trojan applets in a smart card," *Journal in computer virology*, vol. 6, no. 4, pp. 343–351, 2010.
[5] G. Bouffard, J. Iguchi-Cartingy, and J.-L. Lanet, "Combined software and hardware attacks on the java card control flow," *CARDIS*, september 2011.
[6] E. Hubbers and E. Poll, "Transactions and non-atomic API calls in Java Card: specification ambiguity and strange implementation behaviours," Radboud University Nijmegen, Dept. of Computer Science NIII-R0438, 2004.
[7] G. Barbu, H. Thiebeauld, and V. Guerin, "Attacks on java card 3.0 combining fault and logical attacks." in *CARDIS*, ser. Lecture Notes in Computer Science, D. Gollmann, J.-L. Lanet, and J. Iguchi-Cartigny, Eds., vol. 6035. Springer, 2010, pp. 148–163.
[8] E. Vetillard and A. Ferrari, "Combined attacks and countermeasures," *Smart Card Research and Advanced Application*, pp. 133–147, 2010.
[9] T. Razafindralambo, G. Bouffard, and J.-L. Lanet, "A friendly framework for hidding fault enabled virus for Java based smar tcard," in *26th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy DBSEC 2012*, vol. 7371. Springer, 2012, pp. 122–128.
[10] A. Bkakria, G. Bouffard, J. Iguchy-Cartigny, and J.-L. Lanet, "OPAL: an open-source Global Platform Java Library which includes the remote application management over HTTP," *e-smart*, september 2011.