Tampering with Java Card Exceptions
The `Exception` Proves the Rule

Guillaume Barbu[1,2], Philippe Hoogvorst[1] and Guillaume Duc[1]
[1]*Institut Mines-Télécom / Télécom ParisTech, CNRS LTCI, Département COMELEC,*
46 rue Barrault, 75634 Paris Cedex 13, France
[2]*Oberthur Technologies, Innovation Group,*
Parc Scientifique Unitec 1 - Porte 2, 4 allée du Doyen George Brus, 33600 Pessac, France

Keywords: Java Card, Java Exceptions, Software Attacks, Fault Attacks, Combined Attacks.

Abstract: Many publications have studied the various issues concerning Java Cards security regarding software and/or hardware attacks. However, it is surprising to notice that the particular case of exception-related mechanisms has not been tackled yet in the literature. In this article, we fill this gap by proposing several attacks against Java Card platforms based on both exception handling and exception throwing. In addition, this study allows us to point out that a weakness known by the web-oriented Java community for more than a decade still passes the different steps of the state-of-the-art Java Card application deployment process (namely conversion and verification). This appears all the more important as the Java Card 3 *Connected Edition* specifications have started to bridge the gap between the two worlds that are Java Cards and Java web services.

1 INTRODUCTION

The Java Card technology is, as of today, the world's leading technology in the smart card field. This leadership comes from the outstanding cost reduction in terms of application deployment it allows. The famous *write once, run everywhere* motto of the Java technology is therefore a key factor for this success.

As part of the Java language, the notion of *exception* appears important. As per (Chen, 2000), *"an exception is an event that disrupts the normal flow of instructions during the execution of a program"*. Indeed, exceptions are originally meant to handle errors that might occur during the program execution. These errors may come from the Java Card Virtual Machine (JCVM) when internal errors occurs (a null pointer, or an access out of the bounds of an array for instance) or from the application itself. The latter case has lead to an evolution of the use of exceptions. Many applications actually use this mechanism as a programmatic trick to redirect the instruction flow to specific treatments in various occasions.

Another important factor, and especially in the smart card field, is the inherent security of the system. This security is mainly achieved by the Java Card language and the Java Card Runtime Environment (JCRE) security properties. The strongly-typed property of the Java Card language for instance ensures that no pointer arithmetic is used in a Java Card application and that objects behave according to a given contract defined by their Java class, superclasses and interfaces. The JCRE enforces other security rules, ensuring, for instance, the isolation between the different applications running on the platform. The complete list of these requirement can be found in the specifications (Gosling et al., 2005; Lindholm and Yellin, 1999; Sun Microsystems Inc., 2009b; Sun Microsystems Inc., 2009c).

The Java Card 3 *Connected Edition* has brought the Java Card platform really close to standard ones, introducing notions and facilities related to standard Java applications and web services in particular. The literature on the security of Java and web services is relatively vast and several studies, security breaches and security guidelines have been published (Dean et al., 1996; Ladue, 1997; Princeton University, Department of Computer Science, Secure Internet Programming Group, ; Last Stage of Delirium Research Group, 2002; Cholakov and Milev, 2005; Hubert et al., 2010; Long et al., 2011; Oaks, 2001; The Open Web Application Security Project (OWASP), 2012a; The Open Web Application Security Project (OWASP), 2012b; Mehta and Sollins, 1998; McGraw and Felten, 2000; Caromel and Vayssière, 2001; Gutterman and Malkhi, 2005; Livshits and Lam, 2005). It appears then all the more interesting to reconsider these studies in the Java Card context. In this article, we focus our study on the security of the exception-related mechanisms of the Java Card platform. This concern comes from the observation that attackers often try to avoid an exception throwing, but rarely to take advantage of it.

This article is organized as follows. Section 2 introduces more precisely the notion of exception and the related mechanisms. In Section 3, we present several new attacks against Java Card platforms taking advantage of the exception-related mechanisms. Section 4 analyses those attacks and outlines their consequences. Finally Section 5 concludes this article.

2 EXCEPTIONS IN JAVA CARD PLATFORMS

This section aims at introducing the notion of exception, the evolution of its use in a program and the related mechanisms on Java Card platforms, namely exception throwing and exception handling. Although we get relatively specific concerning certain properties, we do not intend to give an exhaustive description of the ins and outs of exceptions. Readers wishing to learn more should definitely refer to (Chen, 2000, §6), (Gosling et al., 2005, §11) and (Lindholm and Yellin, 1999, §2.16).

2.1 The Role of Exceptions

As previously stated, exceptions are initially meant to handle errors during the execution of a program. In the following, we show that if this original purpose is still in use today, exceptions have been totally integrated in application's programming and are now widely used for more than ”just” handling errors.

Handling Abnormal Conditions. On one hand, we find the traditional usage of exceptions, that is to say error handling. In this scope, an exception is created when an abnormal behaviour is detected, when the execution of a program is deemed unsuccessful, or when the system detects that pursuing the execution will lead it to halt abruptly for instance. The exception handler is then in charge of handling the abnormal conditions that have led to the exception raising, and in the worst case to properly halt the system.

Handling Particular Conditions. On the other hand, in certain modern programs, exceptions are used as a way to break the control flow of an application and not necessarily because an abnormal condition has occurred. For instance, an exception may be raised by a program when a given condition is satisfied or not, regardless of the potential consequences of this condition. The aim of such exception is only to force a jump and execute specific lines of code that are defined as the exception handler.

Good Practices. Several references (Long et al., 2011; Oaks, 2001; The Open Web Application Security Project (OWASP), 2012a; The Open Web Application Security Project (OWASP), 2012b) detail the importance of properly handling any exception that might occur during the execution of a program. Concerning the Java language, it is particularly important to segregate the different exception types (classes) in different *catch*-blocks in order to handle each and every exception type in an appropriate way. Furthermore, it is also generally advised not to transmit too much information within the exception, as it could be useful to an attacker[1].

2.2 The Syntax of Exception Handling

Java, as several programming languages, defines a syntax associated to exception handling. For that purpose, the Java language allows the definition of *try-catch*-blocks.

Listing 1: Exception syntax in Java and Java Card.

```
try {
  // Code operating a sensitive
  // process that will raise an
  // exception if deemed unsuccessful.
  ...
} catch (ExceptionType1 et1) {
  // The operation has failed in a
  // specific way, handle it
  // accordingly.
  ...
} catch (ExceptionType2 et2) {
  // The operation has failed in
  // another specific way, handle it
  // accordingly.
  ...
} finally {
  // Code executed whether an
  // exception has been thrown or not,
  // caught or not.
  ...
}
```

As described in Listing 1, the programmer has to

[1]The typical example of such a situation is that of an attacker sending SQL requests to a database server and gaining information on the structure of tables thanks to the message contained in the returned exceptions.

write the code that is likely to raise an exception inside a *try*-block and the code that will be executed when a certain type of exception is raised in different *catch*-blocks. In addition, the *finally*-block allows to define a code portion that will be executed whether an exception was raised or not, caught or not.

2.3 The Exception Handler Table in Java Card Binaries

In the binary format, either .CLASS files for standard Java binaries or .CAP files for Java Card binaries, exception handlers are represented for each method into a so-called *handler table*. In a .CAP file, for instance, this table is represented in the method component as specified in (Sun Microsystems, 2006) and described below:

```
method_component {
    u1 tag
    u2 size
    u1 handler_count
    exception_handler_info
       exception_handlers[handler_count]
    method_info methods[]
}
exception_handler_info {
    u2 start_offset
    u2 bitfield {
        bit[1] stop_bit
        bit[15] active_length
    }
    u2 handler_offset
    u2 catch_type_index
}
```

According to the specifications, the handlers should be sorted according to their starting offset in the method's bytecode. For each handler, the exception handler table specifies the start- and end-offset of the handler in the method's bytecode array, as well as the subtype of *Throwable* it handles.
Therefore, when an exception is thrown at some point during the execution of an application, the JCVM is responsible for searching the appropriate exception handler in the table, if any. The exception handler searching is typically done as presented in Algorithm 1.

The present section has introduced the notions and mechanisms related to exceptions and their handling on Java Card platforms. The purpose of the following section is to emphasize how various attacks on these mechanisms can expose the platform, the hosted applications or sensitive data.

Algorithm 1: Exception Handler Searching.

```
input : E the exception being handled
input : H the exception handler table
output: -
1  while more handler left in H do
2      handler ← next handler;
3      if jpc is covered by handler offsets then
4          if handler match E's type then
5              jump to handler;
6              end loop;
7          end
8      end
9  end
10 forward exception to previous frame;
```

3 SOFTWARE, FAULT AND COMBINED ATTACKS: ON THE SECURITY OF EXCEPTIONS AND EXCEPTION HANDLING

Most of time, attackers only care about exceptions because they wish to avoid the particular conditions leading to an exception throwing. In this section we start by introducing the notion of Fault Attack and describing two previous works taking advantage of an exception throwing to mount an attack. Subsequently, we expose the results of our study on the security of exception-related mechanisms. This consists in the description of new attacks grounded on both exception handling and exception throwing.

3.1 Previous Works

The case of exceptions has not been much investigated in the literature related to Java Card security. However we can outline the work of Barbu *et al.* presented at CARDIS 2010 (Barbu et al., 2010). On the other hand, the standard Java security field has shown much more interest for exceptions (Dean et al., 1996; Ladue, 1997; Princeton University, Department of Computer Science, Secure Internet Programming Group, ; Last Stage of Delirium Research Group, 2002; Cholakov and Milev, 2005; Hubert et al., 2010; Long et al., 2011; Oaks, 2001; The Open Web Application Security Project (OWASP), 2012a; The Open Web Application Security Project (OWASP), 2012b). We briefly describe in particular here the work of *Dean et al.* presented as soon as 1996 at the IEEE Symposium on Security and Privacy (Dean et al., 1996). But let us first briefly introduce the notion of Fault Attacks and recall the widely used fault models.

3.1.1 Fault Attacks on Smart Cards

Fault Attacks aim at disturbing the execution of an application. Since the first published attack based on an optical fault injection (Skorobogatov and Anderson, 2002), several means have been studied to provoke errors in Integrated Circuit. Today, the principal means to disrupt a component like a smart card are light beams and electromagnetic pulses (Quisquater and Samyde, 2002). *In fine*, the disturbance could lead to obtain a faulty output or to execute the application with granted privileges (Giraud and Thiebeauld, 2004; Bar-El et al., 2006). Attested fault models consist for instance in stucking a byte to $0x00$, $0xFF$ or a random value, or in disrupting the native code fetching and thus jumping one or several instructions.
Such facilities have been mainly used against embedded cryptosystems, but they actually are a threat to any function implemented on embedded devices. The idea of combining such physical attacks with malicious applications was introduced in (Barbu, 2009). These *Combined Attacks* allow for instance to disobey certain rules stated by the Java Card specifications. The fault injection aims at disrupting mechanism enforcing these rules and the malicious application exploits the provoked breach. Since then, many Combined Attacks have been published to attack several vital points of a Java Card such as the APDU buffer, the application firewall or the operand stack for instance (Vétillard and Ferrari, 2010; Barbu et al., 2010; Barbu et al., 2011; Barbu and Thiebeauld, 2011; Barbu et al., 2012b; Barbu et al., 2012a).

3.1.2 Using an Exception to Attack a Java Card

In (Barbu et al., 2010), the authors propose an attack on a Java Card 3 platform allowing to read and modify the bytecode array of an on-card application. This is achieved thanks to a type confusion involving the newly introduced *java.lang.Class* class of the Java Card 3 Application Programming Interface (Sun Microsystems Inc., 2009a). In this work, the authors explain that the *java.lang.ClassCastException* thrown by the virtual machine when an incorrect typecasting is executed can be used by an attacker. The aim is then to detect the execution of the exception throwing by monitoring the power consumption of the card in order to determine the precise moment when to apply the laser pulse to disrupt the virtual machine's execution during its type checking.

3.1.3 Using an Exception against a Java Client

In (Dean et al., 1996), the authors describe how the exception throwing can lead to security breaches allowing a privilege escalation on the client-side through a web browser (Netscape Navigator for instance) executing a Java applet. The article is based on the fact that the incomplete execution of a class constructor may lead to an object instance partially initialized. In particular, they exhibit the example of the abstract class *ClassLoader* from Sun's Java Runtime Environment (JRE). They provide a custom class loader extending this abstract class (cf. Listing 2).

Listing 2: The incomplete class loader.

```
class CL extends ClassLoader {
    CL() {
        try { super(); }
        catch (Exception e) {}
    }
}
```

The partial initialization of the class loader leads to a type confusion and eventually to a privilege elevation for the attacker's application. The attack described in this work is then completely based on the exception throwing. This breach was then concealed with updates of both the Netscape Navigator and the Java Development Kit, respectively with versions 2.02 and 1.02.

3.2 Attacks on the Exception Handler Research

In the following, we propose several ways an attacker can take benefits from the Algorithm 1, searching the proper handler for a given exception. We describe first a new attack based on a fault injection that can be either combined with a malicious application or not depending on the attack scenario. Then we introduce a software-only attack using an incorrect handler table within a .CAP file.

3.2.1 Jumping in the Wrong Exception Handler

In the attack presented here, we consider a fault injection targeting one of the conditional branch instructions of Algorithm 1 (*i.e.* lines 3 or 4). The physical perturbation consists in jumping a given (set of) instruction(s) by disrupting the code fetching in the implementation of this algorithm in the Java Card Virtual Machine. Such a fault model is widely accepted in the literature (Skorobogatov and Anderson, 2002; Giraud and Thiebeauld, 2004; Bar-El et al., 2006) and was put into practice in (Barbu et al., 2010) against the `checkcast` implementation.

Provided the attacker meets success with the fault injection, she can then force the jump in an exception handler that was meant for another type of exception, or for the same type of exception but at a different time (*i.e.* covering another part of the code). The following code sample (Listing 3) is definitely far-fetched, but is a perfect illustration of the type of code likely to be targeted by such an attack.

Listing 3: Attacked code example.

```
try {
  // MyOwnPIN.check method throws a
  // TryAgainException when an invalid
  // PIN is submitted.
  if (myOwnPin.check(submitted,0,4)) {
    executePrivilegeMethod();
  }
} catch (NullPointerException npe) {
  // myOwnPin is not yet initialized,
  // do it and return
  myOwnPin = new MyOwnPIN(TRY_LIMIT,
                 PIN_SIZE);
  myOwnPin.update(PIN_ARRAY,PIN_OFFSET,
                 PIN_LENGTH);
  return PIN_NOT_INIT;
} catch (TryAgainException tae) {
  processInvalidPINSubmittedEvent();
  return INVALID_PIN;
}
```

In this particular case, a successful perturbation allows to reinitialize the PIN object, by executing the code in the *catch(NullPointerException)*-block. An attacker reaching a good fault injection repeatability is then likely to brute-force the PIN, although it is supposed to be protected by a counter limiting the number of tries and decremented within the *MyOwnPIN.check* method. Note that in the general case, by forcing a jump in the wrong handler, the attacker always creates a type confusion between the actual exception type and the exception type that is caught. The consequences of the execution of the wrong *catch*-block are on the other hand totally dependent on the application.

3.2.2 Handler Table Corruption

As described in Section 2.3, the .CAP file contains a handler table defining where to jump when a given exception occurs at a given offset in the bytecode array. We explore here the possibility for an attacker to achieve the same kind of attack as just described by the mean of a sole .CAP file manipulation.

When an exception is thrown, Algorithm 1 leads to jump at the offset defined within the handler table. It is therefore obvious that corrupting the handler table will eventually result in a jump at an incorrect offset in the bytecode array. However, before being able to execute the application on-card, the attacker would most likely have to make it pass the bytecode verification. It is then necessary to study how the bytecode verifier ensures the correctness of the handler table. The rules used by the verifier can be deduced from the rules defined in the JCVM specification. These rules are the following:

- handlers are sorted according to their *start-offset*,
- handlers do not "partially intersect", *i.e.* they are either disjoint or nested,
- handlers point to valid offsets in the current method,
- the stop-bit defines when to stop searching for other handlers.

We then tested several modifications of the handler table versus both the Java Card converter and the bytecode verifier present in the JCDK 3.0.4. It turns out that only one manipulation was not rejected. This manipulation consists in the manipulation of the end-offset of a *try*-block so that it englobes the handler of the associated *finally*-block. This is depicted in the .JCA file obtained from the conversion of the applet (after the .CLASS file has been modified) in Listing 4. Note that a modification of the *bit-stop* of the finally handler was also necessary.

Consequently, when an exception is thrown within the *try*-block, it is handled by the *finally*-block which in turn re-throws the exception. This exception is subsequently handled by the same *finally*-block again, *etc*... As a matter of fact, the applet is stuck in an infinite exception *throw-and-catch* loop. Indeed a binary file manipulation can allow to break the infinite loop by using a counter incremented in the loop and conditioning the exception throwing. Also, the *finally*-block can be manipulated in order to use the local variable created by the compiler to store the exception before throwing it at the end of the block. We believe this last point should be very tricky to handle for the bytecode verifier, although we have not been able to find any explicit breach.

3.3 Attacks on Exception Throwing

In this section, we focus our interest on the exception throwing itself. We can therefore isolate two different kind of attacks whether we intend to force an ill-behaviour of this mechanism or simply to bypass it. The following introduces attacks in both categories.

3.3.1 Throwing a Non-throwable

In (Barbu et al., 2011), the authors studied the possible impacts of a perturbation of the values pushed onto the operand stack of a Java Card platform. Some

Listing 4: JCA file after modification.

```
.method public testException()V 8 {
     .stack 2; .locals 2;
 L0: aconst_null;
     astore_1;
 L1: aload_1;
     aload_1;
     invokevirtual 11;
     // equals(Ljava/lang/Object;)Z
     pop;
     new 12;
     // java/lang/Object
     dup;
     invokespecial 3;
     // java/lang/Object.<init>()V
     astore_1;
     goto L4;
 L2: astore_2;
     // local variable created by the
     // compiler to store the
     // exception.
 L3: new 12;
     // java/lang/Object
     dup;
     invokespecial 3;
     // java/lang/Object.<init>()V
     astore_1;
     aload_2;
     athrow;
 L4: return;
 .exceptionTable {
     // start_block, end_block,
     // handler_block, catch_type_index
     L1, L4, L2, 0;
     // handler covers finally-block's
     // athrow!
     L2, L3, L2, 0;
 }
}
```

of their use cases are based on the attacker's instantiating as many objects of a given class $\mathcal{C}$ as possible in order to enhance the probability that a random error affecting an object's reference transform it into one of the reference of an instance of class $\mathcal{C}$.

We tackle here an issue that was not treated in their article: the perturbation of a thrown exception's reference. The possible consequences are twofold depending on the nature of class $\mathcal{C}$:

1. $\mathcal{C}$ extends the API's class *Throwable*. Then the execution will continue in the wrong exception handler, provoking, for sure, a type confusion between the two exception classes and potentially bypassing some exception-dependent operations, including security-related ones.

2. $\mathcal{C}$ does not extend *Throwable*. In this case, the JCVM should not find an exception handler matching the type of the exception. The exception should therefore not be caught and cause the system to halt. However, considering a malicious application, the attacker may have created a *finally*-block taking this possibility into account, which would at least provoke a type confusion between two object instances.

Indeed, in the latter case, even jumping in the *finally*-block is not obvious on all platforms. The *finally*-block being meant to handle any kind of exceptions, it is associated in the application's binary file to a type referred to as *ANY* in the specifications. Yet, every platform is responsible for testing that the object thrown can be casted to the type *Throwable* before jumping in the *finally* handler.

3.3.2 Prevent Exception Throwing

Another interesting option from an attacker's point of view is simply to prevent the throwing of an exception. In the context of Java Cards, this can be achieved on two different circonstances, by two different means.

The first one concerns the case where the exception is thrown by the JCRE. The typical example for such a situation is when an application invokes a virtual method on an object that turns out to be null. As per the specifications, the JCRE then throws a *NullPointerException*. The code responsible for the exception raising is obviously part of the JCRE/JCVM implementation, we can therefore expect it to be written in native language (i.e. in the card's assembly language). Therefore, an attacker willing to skip the exception throwing will have to disrupt the execution of native code, which is generally done by perturbing the fetch of the code either in the ROM or the NVM.

The second one concerns the case where the exception is thrown by a Java Card application. In this case, it is the bytecode instruction *athrow* which is responsible for the exception throwing. An attacker can then work two different attack angles. The first one is the same as previously stated. Since the *athrow* is also a part of the JCVM implementation, a similar attack on the appropriate code fetching shall allow to bypass the execution throwing. The second angle consists in an attack of the bytecode instruction reading in the application's bytecode array. This bytecode is basically a byte read and processed by the JCVM's interpreter. A fault attack stucking this byte to 0 will then force the execution of the *nop* instruction, on platforms using the standard instruction set, and therefore bypass the exception throwing. This is one of the basic examples introducing mutant application in (Séré et al., 2010).

3.3.3 The Incomplete Object Initialization on Java Card

We explore here the possibility to somehow adapt the incomplete object initialization attack described in Section 3.1 on a Java Card. The point is that according to the 2nd edition of the Java Virtual Machine specification, the class file verification process should prevent such a situation from occurring, as it specifies:

> ”A valid instruction sequence must not have an uninitialized object on the operand stack or in a local variable during a backwards branch, or in a local variable in code protected by an exception handler or a finally clause.” (Lindholm and Yellin, 1999, §4.9.4)

In addition, the 3rd edition of the Java Language specification specifies that during an object initialization calling a superclass construcor:

> ”If that constructor invocation completes abruptly, then this procedure [object initialization] completes abruptly for the same reason.” (Gosling et al., 2005, §12.5)

The possibility to load an application containing the code depicted in Listing 2 is then subject to question on recent platforms.

Is it Possible to Load such an Application on a Java Card? We consider then the following constructor method, for a class extending a super-class whose constructor throws a particular exception, say *MyException* in our case (Listing 5).

Listing 5: Potential incomplete initialization.

```
try {
  super();        // throws MyException
  authRequired = true;
}
catch (MyException me) {
  // Initialization is not complete...
  // authRequired = false, the
  // default value for boolean.
}
```

The fact is that trying to compile this code with the Java compiler provided with the Java Development Kit version 1.6.0_18-b07 result in the following error:

```
call to super must be first statement in
constructor.
```

We then assume that this error is due to the restrictions quoted above since the call to the superclass's constructor is indeed the first statement in our constructor. Getting a valid class actually representing this constructor requires then the compilation of a slightly different piece of code and a little .CLASS file tweaking. The content of the two .CLASS files is given in Listings 6 and 7.

Listing 6: Original constructor class.

```
public void <init>()
         throws MyException {
 // Call super constructor
 aload_0
 invokespecial void MySClass.<init>()
try-block_start(app.MyException)_4:
 aload_0
 invokevirtual void MyClass.notSuper()
 aload_0
 iconst_1
 putfield boolean MyClass.authRequired
try-block_end(app.MyException)_13:
 goto label_17
exception_handler(app.MyException)_16:
 astore_1
label_17:
 return
}
```

Listing 7: Tweaked constructor class.

```
public void <init>() {
try-block_start(app.MyException)_0:
 // Call super constructor
 aload_0
 invokevirtual void MySClass.<init>()
 aload_0
 iconst_1
 putfield boolean MyClass.authRequired
try-block_end(app.MyException)_9:
 goto label_13
exception_handler(app.MyException)_12:
 astore_1
label_13:
 return
}
```

Afterward, we successfully pass both the Java Card converter and bytecode verifier that are provided with the most recent Oracle's development kit (Java Card Classic Edition 3.0.4 Development Kit). These two operations yields a verified .CAP file, ready to be loaded and installed on-card, proving that the verifier does not reject classes with potential incompletely initialized objects. We can therefore be pretty confident on the success of the loading of an application with the same behaviour on a Java Card 3 Connected Edition endowed with an on-card bytecode verifier although the lack of publicly available cards prevents us from testing it.

4 CONSEQUENCES AND ANALYSIS

The previous section has shown that several possibilities are offered to a potential attacker to take advantage of the exception-related mechanisms of the system. We intend here to analyse the consequences of such attacks if they were to be achieved on the field.

4.1 Consequences

The consequences of the potential attacks described in the previous section are various indeed. Concerning the incomplete object initialization, the assets that might be targeted by such attacks are still to be precisely identified. Unlike standard Java, the Java Card 3 *Connected Edition* does not support user-defined class loaders, nor the *Object.finalize()* method evoked in (Hubert et al., 2010) to extend *Dean et al.*'s work (Dean et al., 1996). Consequently, this seminal work cannot be transferred in the Java Card world as is. Furthermore, applications taking advantage of all the facilities offered by the Java Card 3 *Connected Edition* standard are not really widespread yet. However the example of the original attack is explicit and should encourage every developer to keep this threat in mind.

Regarding the other attacks we have introduced in this article, the consequences can be divided into two categories. On one hand, the consequences of a type confusion covers the access to private fields or to data out of the bounds of an array, the jump of access control checks, or even self-modifying applications in certain contexts. Type confusion is indeed the core of several attacks against Java and Java Card platforms (Govindavajhala and Appel, 2003; Witteman, 2003; Mostowski and Poll, 2008; Séré et al., 2009; Barbu et al., 2010). On the other hand, the execution flow disruption has been less studied but can lead to critical consequences. First, since it can be used to provoke a type confusion, the same results can be expected. Second, preventing an exception from being thrown can have even more dangerous consequences. We can consider for instance the case of an access across the application firewall enforced by the JCRE that would not throw a *SecurityException*, although it has been duely detected, which is an example given in (Vétillard and Ferrari, 2010).

4.2 The `Exception` Proves the Rule

We outline in this work the potential weakness introduction coming with the evolution of a system. For instance, the integration of the Java Card in the web ecosystem is a topic of interest from a security point of view.

Although the Java Card platform can definitely be considered as an inherently secured platform, it is still potentially vulnerable to attacks. Both software and combined attacks are then likely to threaten the integrity of the platform and to a certain extent of other applications. The implementors of the embedded JCREs, as well as application developers must therefore take this kind of threat into account when coding and comply to the security guidelines. The various attacks we have introduced along this article against exception-related mechanisms allows us to highlight this state of fact.

5 CONCLUSIONS

In this article, we have given an overview of the potential attacks that can be built upon, and around, the exception-related mechanisms of the Java Card technology. In particular we have exposed that attacks on these mechanisms are likely to bypass certain operations contained in *catch*-blocks and to break the type safety property of the Java Card Virtual Machine. In addition, this study has allowed us to exhibit a residual weakness regarding the incomplete initialization of objects which had already been pointed out in the standard Java web applet context with serious consequences on the client side executing the applet. This leads to conclude on the necessity of establishing a link between the various security fields. This necessity is all the more urgent with the introduction of the Java Card 3 *Connected Edition*, bridging the gap between the Java Card world and the Java web-related world.

REFERENCES

Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., and Whelan, C. (2006). The Sorcerer's Apprentice Guide to Fault Attacks. *IEEE*, 94(2):370–382.

Barbu, G. (2009). Fault Attacks on Java Card 3 Virtual Machine. In *e-Smart'09*.

Barbu, G., Duc, G., and Hoogvorst, P. (2011). Java Card Operand Stack: Fault Attacks, Combined Attacks and Countermeasures. In (Prouff, 2011), pages 297–313.

Barbu, G., Giraud, C., and Guerin, V. (2012a). Embedded Eavesdropping on Java Card. In *Proceedings of the IFIP International Information Security and Privacy Conference 2012 – SEC 2012*. Springer Verlag. to be published.

Barbu, G., Hoogvorst, P., and Duc, G. (2012b). Application-Replay Attack on Java Cards: When the

Garbage Collector Gets Confused. In Barthe, G. and Livshits, B., editors, *International Symposium on Engineering Secure Software and Systems – ESSoS 2012*, Lecture Notes in Computer Science. Springer.

Barbu, G. and Thiebeauld, H. (2011). Synchronized Attacks on Multithreaded Systems - Application to Java Card 3.0 -. In (Prouff, 2011), pages 18–33.

Barbu, G., Thiebeauld, H., and Guerin, V. (2010). Attacks on Java Card 3.0 Combining Fault and Logical Attacks. In (Gollmann and Lanet, 2010), pages 148–163.

Caromel, D. and Vayssière, J. (2001). Reflection on MOPs, Components, and Java Security. In *Proceedings of the Engineering C of Object-Oriented Programs (ECOOP)*, volume 2072 of *LNCS*. Springer-Verlag.

Chen, Z. (2000). *Java Card Technology for Smart Cards, Architecture and Programmer's Guide*. Addison-Wesley.

Cholakov, N. and Milev, D. (2005). The Evolution of the Java Security Model. In *Proceedings of the International Conference on Computer Systems and Technologies (CompSysTech'2005)*.

Dean, D., Felten, E. W., and Wallach, D. S. (1996). Java Security: From HotJava to Netscape and Beyond. In *Proceedings of the IEEE Symposium on Security and Privacy*.

Giraud, C. and Thiebeauld, H. (2004). A Survey on Fault Attacks. In Quisquater, J.-J., Paradinas, P., Deswarte, Y., and Kalam, A. E., editors, *Smart Card Research and Advanced Applications VI – CARDIS 2004*, pages 159–176. Kluwer Academic Publishers.

Gollmann, D. and Lanet, J.-L., editors (2010). volume 6035 of *Lecture Notes in Computer Science*. Springer.

Gosling, J., Joy, B., Steele, G., and Bracha, G. (2005). *The Java Language Specification*. Addiosn-Wesley, 3rd edition.

Govindavajhala, S. and Appel, A. (2003). Using Memory Errors to Attack a Virtual Machine. In *IEEE Symposium on Security and Privacy*, pages 154–165. IEEE Computer Society.

Gutterman, Z. and Malkhi, D. (2005). Hold Your Sessions: An Attack on Java Session-Id Generation. In *Proceedings of the Cryptographer's Track at the RSA Conference (CT-RSA)*, LNCS. Springer.

Hubert, L., Jensen, T., Monfort, V., and Pichardie, D. (2010). Enforcing Secure Object Initialization in Java. In *Proceedings of the European Symposium on Research in Computer Securiy*, ESORICS'10, pages 101–115. Springer-Verlag.

Ladue, M. D. (1997). When Java was One: Threats from Hostile Bytecode. In *Proceedings of the 20th National Information Systems Security Conference*, pages 104–115.

Last Stage of Delirium Research Group (2002). Java and Java Virtual Machine Security Vulnerabilities and their Exploitation Techniques. In *BlackHat Conference*.

Lindholm, T. and Yellin, F. (1999). *Java Virtual Machine Specification*. Addison-Wesley, Inc., 2nd edition.

Livshits, B. and Lam, M. S. (2005). Finding Security Vulnerabilities in Java Applications with Static Analysis. Technical report, USENIX.

Long, F., Mohlndra, D., Seacord, R. C., Sutherland, D. F., and Svoboda, D. (2011). *The CERT Oracle Secure Coding Standard for Java*. Carnegie Mellon Software Engineering Institue (SEI) series. Addison-Wesley.

McGraw, G. and Felten, E. W. (2000). *Getting Down to Business with Mobile Code*. John Wiley & Sons.

Mehta, N. V. and Sollins, K. R. (1998). Expanding and Extending the Security Features of Java. In *Proceedings of the 7th USENIX Security Symposium*.

Mostowski, W. and Poll, E. (2008). Malicious Code on Java Card Smartcards: Attacks and Countermeasures. In Grimaud, G. and Standaert, F.-X., editors, *Smart Card Research and Advanced Applications, 8th International Conference – CARDIS 2008*, volume 5189 of *Lecture Notes in Computer Science*, pages 1–16. Springer.

Oaks, S. (2001). *Java Security*. O'Reilly, second edition.

Princeton University, Department of Computer Science, Secure Internet Programming Group. Reports on Security Flaws in Commercial Available Softwares.

Prouff, E., editor (2011). volume 7079 of *Lecture Notes in Computer Science*. Springer.

Quisquater, J.-J. and Samyde, D. (2002). Eddy Current for Magnetic Analysis with Active Sensor. In *e-Smart 2002*.

Séré, A. A. K., Iguchi-Cartigny, J., and Lanet, J.-L. (2009). Automatic Detection of Fault Attack and Countermeasures. In *Proceedings of the 4th Workshop on Embedded Systems Security*, WESS '09, pages 1–7.

Séré, A. A. K., Iguchi-Cartigny, J., and Lanet, J.-L. (2010). Checking the Paths to Identify Mutant Application on Embedded Systems. In *FGIT*, pages 459–468.

Skorobogatov, S. and Anderson, R. (2002). Optical Fault Induction Attack. In Kaliski Jr., B., Koç, Ç., and Paar, C., editors, *Cryptographic Hardware and Embedded Systems – CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 2–12. Springer.

Sun Microsystems (2006). Virtual Machine Specification – Java Card$^{\text{TM}}$ Plateform, Version 2.2.2.

Sun Microsystems Inc. (2009a). Application Programming Interface, Java Card Platform, Version 3.0.1 Connected Edition.

Sun Microsystems Inc. (2009b). Runtime Environment Specification, Java Card Platform, Version 3.0.1 Connected Edition.

Sun Microsystems Inc. (2009c). Virtual Machine Specification – Java Card Plateform, Version 3.0.1.

The Open Web Application Security Project (OWASP) (2012a). Information Leakage.

The Open Web Application Security Project (OWASP) (2012b). Uncaught Exceptions.

Vétillard, E. and Ferrari, A. (2010). Combined Attacks and Countermeasures. In (Gollmann and Lanet, 2010), pages 133–147.

Witteman, M. (2003). Java Card Security. In *Information Security Bulletin*, volume 8, pages 291–298.