



## Java Card Security

Marc Witteman

### Introduction

Java Card is a new, but fast growing technology that enhances the world of smart cards with a whole set of exciting new possibilities. Until a few years ago all smart card manufacturers had their own proprietary operating systems. Application developers faced considerable difficulties and costs to get their application to run on multiple platforms. Due to the success of Java in the desktop world, as a language and as a platform, it seemed an attractive idea to design a small Java-based operating system that could become a common factor for all smart card platforms. Collaboration between SUN and a group of smart card manufacturers resulted in the birth of Java Card in 1997.

The most important innovations of Java Card are:

- Interoperable: verified applets run on any Java Card
- Multi-application: multiple applets can co-exist
- Dynamic: new applets can be added post-issuance
- Secure: Java's inherent security is enhanced with dedicated concepts

Today a large quantity of newly produced smart cards is equipped with a Java Card operating system, and card issuers start using the possibilities offered by this technology. The financial and telecom markets are the biggest application areas.

Banks use Java Card applets for financial services like e-purses, debit/credit schemes and loyalty applications. A consortium of banks even developed a complete framework called Global Platform for card and application management. This industry standard is quickly becoming the de facto standard in the smart card payment world.

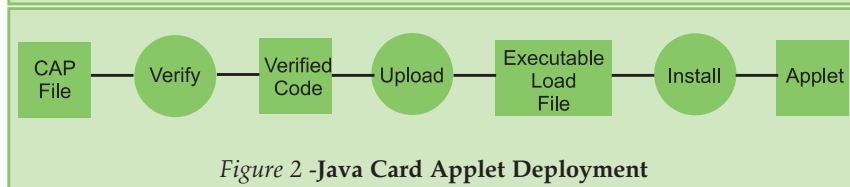
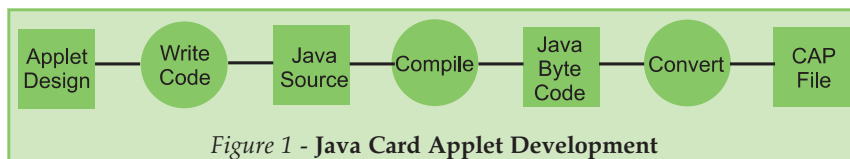
Mobile telecom operators use SIMs running on Java Card for infotainment and network optimisation. Additional standards have been developed for remote management and event dispatching amongst concurrent applets.

This article studies security aspects of the Java Card technology and tries to identify its shortcomings. Three case studies illustrate the impact of various threats. Finally some ideas are presented to counteract the threats and apply this technology in a secure way.

### Basic Principles

Although Java Card may look like just another Java flavour it differs from standard Java in several aspects:

- Language features



- Runtime Environment (Virtual Machine and API)
- Operational processes

The deviations relate to the resource limitations, security requirements and also processing characteristics of smart cards.

The Java Card language is a subset of standard Java. Several language features are omitted, mostly because of resource limitations. Removed features include large primitive data types (such as floats) and multidimensional arrays.

The Java Card Runtime Environment is totally different from that of standard Java. In general none of the Java core API classes are supported, with the exception of a few basic classes that are supported in a limited form. Important unsupported features include dynamic class loading, security manager, threading, cloning and garbage collection. New features are: persistent and transient objects, atomic operations and the applet firewall. Furthermore several classes for security services like authentication and encryption are included.

The operational process can probably be best understood in view of the applet life cycle. The applet life cycle is split in a development and a deployment life cycle.

Developers can create applets independently from any existing hardware or smart card brand. Applets undergo several transformations during development (see Figure 1). The Java source code is compiled just like in the standard Java world. An additional step which is not required in standard Java is the conversion to a CAP (converted applet) file. This is actually a JAR file containing executable binary code of all classes from one package. At this stage the applet development is finished and it can be handed over to a card issuer for deployment.

The first thing a card issuer (or its controlling authority) would do is to verify the correctness of the CAP file (see Figure 2). This is an important step as a Java Card does not have to do on-board byte code verification for performance reasons. Next the verified code can be uploaded to the target Java cards. This can be done pre-issuance during card initialisation, or post-issuance

via a suitable remote management protocol. As a last step the applet needs to be installed, which can be seen as activation before first use.

An important aspect of this process is the fact that card applets are uploaded instead of downloaded: a controlling authority decides to move code to (a large batch) of cards. In the standard Java world applets are down-

loaded (provoked) by the user, who by his behaviour has more control over the content of his Java implementation.

Note that Java Card applets are resident. Once they are downloaded they remain in non-volatile memory until they are explicitly removed.

From a security point of view the most important benefits of Java Card concepts are:

- Removal of risky features like dynamic class loading and threading: simplifies maintaining type safety and avoids security problems due to concurrent thread interactions.
- Introduction of the applet firewall: default applet isolation and controlled interactions.
- Support of atomic operations: guarantee consistency of related data fields.

It is important to realise that the new concepts introduce particular risks associated with a correct (secure) implementation of these mechanisms.

A more detailed introduction to the Java Card technology is outside the scope of this article, and can be found in *Java Card Technology for Smart Cards*[1]. An introduction to smart card security can be found in *Advances in Smart Card Security*[2].

## Security Concepts

Four aspects related to Java Card technology deserve a deeper analysis because they play a vital role in the platform security:

1. CAP Verification
2. Applet loading
3. Firewall
4. Atomicity

### CAP Verification

A CAP file is a binary extract of the original Java code that is suitable for the limited smart card resources due to its small footprint. Unfortunately this small representation has also lost some of the inherent Java language security features. It would be possible for a malicious developer to modify a CAP file such that it violates the Java

language constraints and threatens the security of the hosting platform. It is therefore important to verify that a CAP file still plays by the rules. A CAP file verifier checks that the code is both internally and externally (references to other CAP files) consistent.

As verification is not an easy task it is not required for Java cards to perform this verification on board, but it is intended to be done off-card, just before applet loading. An off-card verifier checks the consistency of individual CAP files and external references are checked against so called EXP (export) files. These files contain prototypes of the imported classes and their methods (similar to .h header files in C), and are produced during CAP file generation. Thus, off-card CAP file verification requires not only a CAP file but also EXP files for all imported classes. Although not required by the specifications some Java Card manufacturers are moving towards on-card CAP verification as this will contribute to a higher level of security.

An applet that has passed the verification step is called a *Verified Applet*, while an applet that has failed the verification is called an *Ill-formed Applet*. It is important to realise that Java Card virtual machine implementations expect applets to be verified and conform to the Java Card language. Java Cards are not required to withstand attacks from ill-formed applets. It is therefore an absolute requirement for Java Card security that all applets are verified.

### Applet Loading

The Java Card specification does not prescribe a loading process, but suggests that a dedicated applet (installer applet) handles this task. SUN's Java Card development kit contains a simple installer applet without cryptographic support. This installer can be used for demonstration purposes or pre-issuance applet loading, but is not suitable for post-issuance applet loading.

*Global Platform* defines an on-card entity called the *Card Manager* capable of performing more sophisticated loading. Its loading process is also used for GSM Java Card SIMs. The Card Manager implements a secure channel protocol that provides cryptographic services like encryption and authentication to support secure applet loading.

The security of the loading process is paramount. If an adversary would succeed in bypassing the loading process security he would be able to load an ill-formed applet on the card and successively break the security of the other applets and the platform.

### Firewall

The applet firewall takes care of applet isolation and controlled interaction. By default, applets cannot access members belonging to other applets, even if they are public. Applets may want

to share some data or methods though. The classical example involves the payment and loyalty applications: upon completion of a payment some 'points' must be added to the loyalty balance.

The firewall provides an interface for a client applet to request 'sharing' to a server applet. The server applet can grant or deny this access (possibly including client authentication). If the sharing is allowed the client can access public methods from the server that have been exported through a shareable interface.

This sharing procedure performs real time linking of methods. Client and server could possibly use different versions of the sharing interface, leading to type confusion. Therefore the firewall must take precautions to enforce binary compatibility between client and server applets.

### Atomicity

Smart cards often operate in unfriendly environments outside the control of the card issuer. Unfavourable conditions may arise with regard to temperatures, vibrations, humidity, etc. Also, a card holder can at any moment tear the card away from a card accepting device (card reader). These circumstances may occasionally interrupt ongoing operations.

Software processes often rely on internal data consistency, and can become severely disordered by power disruption. Consider for example an electronic purse that uses a variable 'balance' to hold the purse value and a variable 'state' to hold the transaction state. Suppose that power would be disrupted after updating 'balance' but before setting 'state', then there would be no evidence that the transaction was completed and the purse value would be incorrectly interpreted.

Java Card introduces a transaction mechanism that guarantees atomicity. It makes sure that either all operations or no operation within the transaction is completed. A commit operation at the end of the transaction confirms the completion of the previous operations. If the transaction is aborted (by power interruption or software abort), the mechanism makes sure that all previous operations within the transaction are reversed. In this way it is possible to maintain the internal consistency of related data.

A consequence of the transaction mechanism is that objects which were allocated within an aborted transaction are to be deleted and their references shall be reset.

### Threats

Operational Java Cards may hold applets from various suppliers. It is very well possible that applets of competing applet providers co-exist, e.g. multiple loyalty schemes. Applet providers may have different security standards. A financial applet may be developed under strict security conditions while an entertainment applet may be

## SMARTCARD SECURITY

developed in a completely unprotected environment. For all those reasons applet providers and card issuers may be concerned about the operational security of their applet or platform.

Applets may exhibit the following types of undesirable behaviour (in increasing order of severity):

1. Perform harmless but annoying behaviour
2. Crash the card (temporarily or permanently)
3. Engage in unauthorised external behaviour (by using external resources)
4. Expose user confidentiality
5. Attack other applets or the platform

It is important to realise that Java Card does not (or hardly ever) address the 'mild' risks related to the first four of the types of behaviour listed above. E.g., it is trivial for an applet to use regular Java Card features to mute a card temporarily by invoking an endless loop. Card issuers need to address these issues.

Java Card does address the more severe risks of a malicious applet attacking other applets or the platform. The features in previous sections all contribute to this defence. It is not obvious that a Java Card implementation fully mitigates this threat though.

With regard to undesirable behaviour three basic types of threats can be distinguished:

- Verified applet abuses features: applet abuses regular Java Card features to perform undesirable, or even harmful, behaviour
- Verified applet exploits bug: an applet exploits an implementation bug in the platform, or a security loophole in the Java Card specifications, to attack other applets or the platform
- Ill-formed applet compromises platform: unverified applet succeeds in loading onto the platform and attacks other applets or the platform

The following case studies illustrate the impact of each of these threats.

### Case Study: Feature Abuse

Imagine a GSM SIM equipped with a traffic info applet. The applet can help motorists to avoid traffic congestion by querying a data base via their GSM handset. The applet is provided by a third party to the network operator and was developed in an informal environment without standardised security measures. One of the applet programmers decided to include a tiny malicious code fragment (Trojan) within the otherwise very useful applet...

As soon as the mobile subscriber requests service from the traffic info applet he will have to enter the road number first (Figure 3).



Figure 3

Figure 4

Then, to the occasional surprise of the subscriber, the Trojan intervenes, and pretends a SIM problem. It reports a SIM error and requests the subscriber to re-enter his SIM PIN code (Figure 4). The unsuspecting subscriber seems to recognise the

intervention as a 'regular' problem with GSM handsets and enters his PIN obediently. The Trojan subsequently sends out an SMS secretly leaking the PIN code to a third party who can take advantage of this subscriber secret.

This Trojan is actually a piece of perfectly legal Java code that does not have to take more than 10 lines in an applet of a few hundred lines of code. A malicious programmer, who may be serving another purpose than just the business of the application provider, can easily insert it. A brief applet review might even not recognise this piece of code as a malicious Trojan.

### Case Study: Firewall Bug Exploit

This case study shows how type confusion can break Java Card security. Type safety is a cornerstone of Java security, and guarantees that arguments of a method call match the type of a method declaration. This case uses SUN's Java Card reference implementation.

Consider a simple class with one public member:

```
public class ClassA {
    public int x
}
```

And consider another class with one private member:

```
public class ClassB {
    private int y
}
```

Suppose an object implements a method:

```
void doSomething(ClassA a) {
    a.x = 1;
}
```

And suppose this method is called somewhere with an argument of ClassB:

```
..
ClassB b = new ClassB();
doSomething(b);
```

A type safety problem would occur when this code is executed. Method doSomething expects

field `x` to be public and can freely access it. Type confusion arises when object `b` of class `ClassB` is accepted as argument. The method would change the private field `y` now! In general, type confusion leads to a destruction of the inherent language security features.

Java compilers check for type safety and would normally not allow these kind of type confusions. However in relation to applets sharing data via a firewall, type confusion would not always be detected by a compiler.

We now study a case where two sharing applets get illegal access to a part of system memory. The applet firewall is quite restrictive in data exchanged through a shared interface so we will use only primitive type arguments.

Imagine a server applet that implements the following interface:

```
public interface TypeAttackInterface
extends Shareable {
    public void typeAttack (short[] buf);
}
```

The applet is loaded onto a card. Next a client applet is developed, using a slightly different version of the interface:

```
public interface TypeAttackInterface
extends Shareable {
    public void typeAttack (byte[] buf);
}
```

The modified version of the interface is binary incompatible with the original version because the argument types don't match (short array vs. byte array).

Due to the modified interface code a compiler would now expect arguments of method `typeAttack` to be of type byte array. The client applet is loaded on the card and tries to connect to the server applet and invoke the `typeAttack` method on a globally accessible byte array (e.g. the buffer used for I/O with the external world).

A strong firewall implementation would now detect the binary incompatibility and reject the type confusion. The experiment was performed on SUN's reference implementation (version 2.2) and it appeared that type safety is not enforced on this level. A weakness in the implementation is now detected, and the server applet can try to abuse the exploit.

The primitive short type occupies two bytes of storage. A short array thus occupies twice as much memory space as a byte array of the same length. When the server applet looks upon the received byte array as if it were a short array it will have access to twice as much memory space. Obviously the second half of the presumed short array does not belong to the given argument and is actually part of the system memory. This part of memory, that may be in use by other ob-

jects, can contain sensitive information that should not be disclosed.

The author knows variations on this attack that are even more dangerous and can compromise the complete operating system.

Exploits similar to this one are very dangerous as they can be concealed within verified applets. It is essential to this exploit that the CAP files can pass off-card verification if they use binary incompatible EXP files of `TypeAttackInterface`. Card issuers can prevent this type of attack by attentive security officers noticing the binary incompatibility and detecting the exploit.

### Case Study: Type Confusion in an Ill-Formed Applet

Java Card platforms trust that only verified applets are loaded. They are not expected to detect malicious code contained in ill-formed applets. We will show in this case how easy it is for an ill-formed applet to get access to most of the card memory in SUN's Java Card reference implementation. Again we use type confusion.

We first write a correct applet containing a method that copies some data and sends it to the outside world:

```
void processReadMem(APDU apdu)
{
    // get I/O buffer to communicate
    byte[] buffer = apdu.getBuffer();
    byte[] copyFrom = buffer;
    //I/O buffer

    // get incoming data
    apdu.setIncomingAndReceive();

    // copy a short memory block
    Util.arrayCopy(copyFrom, (short)0,
        buffer, (short)4, (short)4);

    // send the data out
    apdu.setOutgoingAndSend((short)0,
        (short)8);
}
```

This method assigns byte array reference `copyFrom` to the I/O buffer. Further it simply copies the first four bytes of incoming data to the next four bytes of the I/O data buffer. Finally it transmits eight bytes of the I/O buffer to the outside world.

Next we design a new and simple class:

```
public class Fake {
    short size;
    public Fake(short s) {
        size = s;
    }
};
```

---

## SMARTCARD SECURITY

The crucial aspect of objects of this class is that it has a short value size on the same place where array objects generally store their length (this can be implementation dependent). If we can assign an array reference to an object of class Fake we let the operating system believe that the length of the referenced array equals the value of the field size.

We now add one line to the applet declaration:

```
private Fake fake = new
Fake((short)0x8000);
    // size set to 32K
```

Next we modify the previous applet method slightly, still keeping it legal:

```
void processReadMem(APDU apdu)
{
    // get I/O buffer to communicate
    byte[] buffer = apdu.getBuffer();
    // create a reference to fake that we
    // will abuse
    Fake tmp = fake;
    // copyFrom will be pointing to fake
    // when changing the CAP file
    byte[] copyFrom = buffer;
                                // I/O buffer
    // get incoming data
    apdu.setIncomingAndReceive();
    // copy requested memory block
    Util.arrayCopy(copyFrom, (short)0,
        buffer, (short)0, (short)100);
    // send the data out
    apdu.setOutgoingAndSend((short)0,
        (short)100);
}
```

To make the attack work we would like to assign the copyFrom reference to fake. This would be disallowed by the Java compiler. Instead we assign a temporary reference tmp to fake and still assign copyFrom to the I/O buffer. Next we compile and convert the code into a CAP file.

The transformation of the verified applet into an ill-formed applet is performed by editing the CAP file. Inside the CAP file we find the assignments of tmp and copyFrom right after each other, and we simply change the copyFrom assignment so that it gets the same value as tmp. Now we have created an ill-formed applet that uses type confusion. The copyFrom reference is associated to an object of class Fake, but the operating system thinks it is an array with a size of 32K.

When we load the CAP file onto the reference implementation it turns out that most of the card

memory can actually be read and dumped through the I/O interface. An experiment showed that 15K of data including sensitive information from all loaded applets could be retrieved.

It is clear that ill-formed applets can wreak havoc on Java Card platforms. It is therefore of the utmost importance that only verified applets be loaded on Java Cards.

### Ensuring Java Card Security

The case studies have shown that various threats can seriously impact Java Card security. We will now investigate what must be done to counteract these threats and be able to apply the Java Card technology in a secure manner.

The following steps should be taken to ensure sufficient security throughout:

- Java Card applet source code reviews
- CAP file verification and code signing
- Ensure loading security
- Challenge platform strength

### Source Code Reviews

Ideally, verified Java Card applets cannot threaten each other. So can we load any verified applet without risk? Unfortunately not because:

- A malicious, or badly designed, applet may still crash the platform, and deny service to other applets.
- A malicious applet may exhibit legal but undesirable behaviour.
- A malicious applet may try to exploit bugs in the platform.
- A benevolent but badly designed applet does not protect its own security (and the associated business).

For all these reasons applet reviews are strongly recommended. Application developers are not always very fond of inspections of their code, but in many cases this is just inevitable. Only in situations where security is no issue for any of the applets, or where the application provider takes full liability over all platform security related calamities it might be acceptable to skip the applet review.

When carrying out source code reviews it is important to have security guidelines. This is a set of rules that applets must adhere to in order to pass the inspection.

Although presently unavailable it might be possible that a dedicated inspection tool becomes available that could perform this task automatically. This would make applet reviewing easier, faster, more acceptable (to the application developers) and probably cheaper too.

**CAP File Verification and Code Signing**

Verification of CAP files is an absolute requirement due to the threat of ill-formed applets. Fortunately, good CAP file verification tools exist so card issuers should just make sure that verification is well embedded within their applet deployment process. It is important that the verification is supported by watertight administrative procedures and that the verification tools are operated by trained professionals to minimise the risk of binary incompatibility exploits. Furthermore the development and improvement of on-card verifiers can significantly increase the security level.

In addition to verification it is a good idea to apply code signing to CAP and EXP files to be transported and/or stored. The data can be protected against unauthorised modification by means of a cryptographic signature. This way application developers and card issuers can be sure that any code files that are exchanged or copied, are intact. Furthermore, a card issuer can be sure that no modifications are made to stored applets after verification.

**Loading Security**

The process of loading applets from the card issuer onto the Java Card is a delicate process. The platform security is at great risk if at any time a modification could be made to the CAP files, or if unverified applets could be loaded.

It is paramount that secure cryptographic algorithms and protocols are used for the transport. Due to continuous research on cryptographic security and increase in the computing power of cracking tools it is advised to periodically re-evaluate the cryptography in use.

The smart cards themselves should also be well protected against unauthorised downloads. Especially the possibility of side channel attacks (see reference [2]) is still a considerable threat for many smartcards.

**Platform Strength**

As shown in this article it is not trivial for Java Card implementations to satisfy all specified security requirements. Since the technology is young it is likely that many platforms still have some vulnerabilities. Also, there is ongoing work on a newer version of the base specifications and extensions that may open up new security challenges.

For card issuers that need to trust their cards it is therefore essential to have their Java Card platforms tested. These tests would take a significant set of deliberately malicious applets that verify all security requirements of the Java Card specifications. Each of the applets (or a combination) is first transported from a test system to a card under test. Then it starts checking a few related security aspects by processing several statements and API calls. Finally the test applet reports a

status back to the test system indicating the result of the test.

**Roles and Responsibilities**

Three parties play major roles in Java Card applications:

1. Applet developer
2. Java Card manufacturer
3. Card issuer

Java Card technology can only be applied in a secure manner if all players understand the associated risks and take the appropriate security measures.

Applet developers need to realise that the security of the target platform depends on the security of their product. They need to standardise on high level operational security measures, to make sure that their products are not only functional, but also secure, robust, reliable and trustworthy.

Card manufacturers need to realise that Java Card security relies heavily on the quality of the hardware and software. They need to protect their products against side channel attacks and be sure that their Java Card implementations conform fully to the standards.

Card issuers need to realise that Java Card technology may not yet be fully mature from a security point of view. They will play a key role in all of the previously mentioned aspects.

**Conclusion**

Java Card is a significant step forward in smart card technology. Security is a key factor throughout the Java Card specifications, but realistic threats remain.

The concept of off-card verification is understandable in the context of limited smart card resources, but it is also more risky than it seems because it puts even higher constraints on loading security.

Three types of threats are identified:

1. Verified applets that abuse regular Java features
2. Verified applets that exploit bugs
3. Ill-formed applets that compromise the platform

The first and second type of threat can be (partly) addressed by means of source code reviews. In addition Java Card platforms need to be tested for susceptibility to exploits.

The third type of threat can be addressed by ensuring CAP file verification and loading security.

Java Card technology can be applied in a secure way when the risks are understood and the in-

---

## SMARTCARD SECURITY

involved parties take responsibility for applying the appropriate security measures.

### Acknowledgement

The author would like to thank Bart Jacobs and Martijn Oostdijk of the Security of Systems group at the Nijmegen Institute of Information and Computing Science for their review and suggestions.

### References

- [1] Zhiqun Chen, Java CardTM Technology for Smart Cards, Addison-Wesley, ISBN 0-201-70329-7, June 2000
- [2] Marc Witteman, Advances in Smart Card Technology, Information Security Bulletin, Issue July 2002
- [3] Gary McGraw & Edward W. Felten, Securing Java, Wiley & Sons, ISBN 0-471-31952-X, 1999
- [4] SUN, Java CardTM 2.2 Application Programming Interface, September 2002
- [5] SUN, Java CardTM 2.2 Runtime Environment (JCRE) Specification, June 2002
- [6] SUN, Java CardTM 2.2 Virtual Machine Specification, June 2002
- [7] SUN, Java CardTM Development Kit, Version 2.2, Binary Release, June 2002
- [8] Global Platform, Card Specification, Version 2.1, June 2001
- [9] ETSI TS 101 476 (GSM 03.19), SIM API for Java CardTM, Version 8.3.0, December 2001
- [10] ETSI TS 101 181 (GSM 03.48), Security mechanisms for SIM application toolkit, Version 8.8.0, December 2001

### About the Author

Marc Witteman got his MSc degree in Electrical Engineering at the Delft University of Technology in the Netherlands. In 1989 he joined KPN (the main Dutch telecoms operator) where he initially worked on GSM development. Later he worked in the field of testing theory, which resulted in a couple of scientific papers and patent applications. In 1994 he moved to the area of smart cards. During 1996 and 1997 he was affiliated with the ETSI standardisation body where he headed a smart card team. In 1997 he moved to the TNO evaluation facility where he became a research leader in smart card security projects. He developed and practised many new smart card evaluation methods. Since 2001 he has been the head of security consulting company Riscure.

He can be reached at [witteman@riscure.com](mailto:witteman@riscure.com).

---

There is *only* one way to get all issues of  
Information Security Bulletin:

**SUBSCRIBING!**

Please use the form in the journal, or visit  
[www.isb-online.net](http://www.isb-online.net)