Testing the Java Card Applet Firewall

Wojciech Mostowski and Erik Poll

Security of Systems (SoS) group
Department of Computing Science
Radboud University Nijmegen
The Netherlands
{woj,erikpoll}@cs.ru.nl

Abstract. In this paper we discuss the methodology and results of testing the Java Card applet firewall mechanism. The main motivation for this work is the complexity of the firewall. Given the complexity, non-compliance of the cards with respect to the official specification is not unlikely. Firewall implementation faults may lead to serious security issues. Although we did not discover any serious problems on our test cards, a few minor specification violations are reported. We only found one specification violation on one card that could be considered unsafe, in that it might introduce a security risk for specific applications.

1 Overview

During investigations of Java Card Virtual Machine security w.r.t. ill-typed bytecode (to be reported in [9]) we considered the Java Card firewall mechanism as one of possible ways to sneak in ill-typed code onto the card. During our tests we realised that the firewall mechanism by itself may offer possibilities to break card security if not implemented correctly. We also realised that the Java Card firewall specification [11, Chapter 6] may not be obvious to interpret at places (although we have to say that we did not find serious ambiguities in the specification). Moreover, there have been substantial changes to the specification along with the introduction of logical channels in version 2.2 of Java Card.

One more reason for taking the firewall specification under the scope is to be able to create a formal model of the firewall for the KeY system [2] to complete our recent work on the verified Java Card API reference implementation [8]. Our earlier work show that the official specifications cannot always be fully relied on when rigorous formalisation is considered. In [5, 4, 7] we reported on ambiguities in the Java Card specification and formalisation difficulties in the context of the Java Card transaction mechanism. In the end we tested some cards and built the formal model partly based on the official specifications (where non-ambiguous) and partly on the actual behaviour of cards (where the specification was not clear or simply underspecified). A natural step before constructing a formal model of the firewall mechanism is to assess the compliance of cards w.r.t. the official specification. This allows to catch the precise semantics of the firewall checks.

All in all, we decided to construct a comprehensive Java Card firewall mechanism test (soon to be available from authors' webpages) to see how faithfully the cards follow the specification and whether there are some non-compliant behaviours that could be exploited to break card security. Although in principle the answer to the latter is "no", we did discover small deviations from the specification. We describe the interesting and relevant parts of the specification together with the test results in Section 4.

1.1 Related Work

Some deficiencies of the Java Card firewall mechanism were already pointed out in 1999 [6, 10]. Then in [1] a system based on higher order logic (the Coq theorem prover) has been proposed for verification of applet isolation properties, i.e. firewall properties. Later in 2004, a type system to statically verify absence of firewall violations was proposed [3]. (While such static verification has the advantage of detecting problems earlier, at compile time, for security having checks at runtime has its advantages, because one does not have to trust code being statically checked.) Since then, we are not aware of any work on either testing or formalising the Java Card applet firewall mechanism. Apart from scientific research, commercial companies engage in Java Card compliance testing. A notable example here is the JCWorkBench tool from Riscure[1] which allows to test cards w.r.t. many aspects of the Java Card platform, including the applet firewall. In fact, we communicated with Riscure during our work, reported our test results to them and also consulted their tests for specific test ideas.

We ran our firewall tests on a set of test Java Cards. In total we tried eight different cards from four manufacturers:

– `A_211` and `A_221` from Manufacturer A, implementing Java Card 2.1.1 and 2.2.1 respectively,
– `B_211`, `B_22`, and `B_221` from Manufacturer B, implementing Java Card 2.1.1, 2.2, and 2.2.1 respectively,
– `C_211A` and `C_211B` from Manufacturer C, both implementing Java Card 2.1.1,
– `D_211` from Manufacturer D, implementing Java Card 2.1.1.

However, it was not possible to run our tests on all of the cards. Cards `C_211A` and `C_211B` refused to install our test applets, most likely due to picky on-card bytecode verifier that prohibits the use of shareable interfaces altogether. Card `D_211` also refused to install our applets for reasons still unclear. Finally, all cards except one (`D_211`) should be considered newer generation cards, regardless of the API version they implement.

The rest of this paper is organised as follows. Section 2 gives background information about the firewall mechanism, Section 3 briefly describes our test suite. In Section 4 we discuss parts of the firewall specification relevant for our test results and point out non-compliant behaviours we discovered in our test cards. Finally, Section 5 concludes the paper.

2 The Firewall Mechanism

The Java Card firewall mechanism [11, Chapter 6] provides a means to guarantee separation of applet data on the card. It prevents an applet from unwanted interference from another applet that could be made possible by it leaking a reference to some internal data structure or exposing some instance field as public. Every applet running on the card has a security context assigned to it, however, more than one applet can reside in one security context. Subsequently, all objects on the card belong to some security context. The context of a given object is assigned during object creation and is inherited from the owner of the object (i.e. the applet). The object access rules are established by the currently active security context. That is, an object can be accessed (e.g. field access, method call) only if the current running context is the same as the security context of the object. Any attempt to access data outside of the context results in a `SecurityException`.

[1] `http://www.riscure.com`

Obviously there are some exceptions from the above mentioned rule. To begin with, the Java Card Runtime Environment (JCRE) has a special system privilege, i.e. it can read data from any security context regardless of any other rules. That is, basically *none* of the access rules apply if the currently running context is the JCRE context. Large parts of the Java Card API code runs in the JCRE context, making them a potential source of security problems, as discussed below.

Then, to enable data exchange between applets, and also accessing system owned data, the mechanism of shareable interfaces and context switching is provided. In short, an applet (the client) can request (through specific API calls) a shareable interface object from another applet (the server). Once the server grants the access, the client receives an object that belongs to the server and that implements some methods defined in some shareable interface (a regular Java interface that extends a special tagging `Shareable` interface). Then the client has the right to invoke methods defined in the interface on the server's object. During every such method invocation a context switch happens. Before the method is called, the current context is that of the client, during the shareable method call the context is switched to the server, and after the method is finished the context is restored back to the client.

To enable the applet to access certain JCRE data, so called system entry points are provided: JCRE owned instances of special classes that can be accessed from any context. In particular, any context can access system owned instances of the `AID` class and the `APDU` object. Again, when methods are invoked on entry point objects, a context switch happens, each such method executes with the JCRE privilege. At this point it is very important for the API to establish that the client does not try to bypass the firewall. In particular, any references that are passed to such a JCRE privileged method do not have to necessarily belong to the client context. Thus, each JCRE method should verify the received references against the client context.

Furthermore, the entry points are divided into permanent and temporary ones. References to permanent entry point object can be stored in instance fields, while temporary cannot. JCRE owned `AID` objects are permanent entry point objects, the `APDU` object is temporary.

Finally, global arrays provide one more way to share data between contexts. Global arrays are accessible from any context. There are only two global arrays in the whole Java Card system: the installation buffer and the `APDU` buffer. Global arrays are also temporary, i.e. their references cannot be stored in instance fields.

On top of the above mentioned rules there are some special cases and exceptions. Most important are:

– Logical channels interact with the firewall mechanism: access to shareable interface object is forbidden if the server applet is selected on another logical channel [11, Section 6.2.8.6].
– Clear-on-deselect arrays are subject to additional access rules as described in [11, Section 6.1.5] and Section 4.5.
– Public static fields do not really belong to any context and thus are accessible from any context. Furthermore, public static methods can be called from any context without causing a context switch [11, Section 6.1.6]. Hence public static fields that are not `final` constitute a potential security risk and should never be used.[2] (Static fields that are `final` are constants, so cannot be tampered with from the outside.)

Obviously, in addition to these firewall rules, the regular Java access rules (private/protected access, etc.) apply during compilation time.

[2] In fact, given that visibility attributes such as public and protected are *not* checked at runtime, *all* static fields that are not `final` constitute a potential security risk. A sensible programming guideline would be to forbid these.

3 The Test Program

As stated already, the main purpose of testing the firewall is to establish cards' compliance w.r.t. the rules we described in the previous section, and assess any security risks that may result from non-compliant behaviour.

To thoroughly test the firewall we have a set of test applets on the card and a dedicated terminal application:

– The terminal application sends simple APDUs to the card, that in turn invoke tests on the card. The card replies with the test result which is then reported by the terminal to the user in human readable form.
– There are three applets on the card and one library package. The library package defines some shareable interfaces that the test applets use and some class definitions that are used in the tests.
  Two of the test applets reside in one package, and the third applet resides in another package. The first package needs two applets to test the compliance w.r.t. two different applets that share the same security context. The third applet obviously serves to test the communication across the firewall.

Our tests methodically cover the firewall rules defined in [11, Chapter 6]. In particular there are tests to check that all firewall access rules are followed, that contexts are switched and reported according to expectations, that entry point objects follow their special rules and preserve security, and that all other special rules we mentioned in the previous section are considered.

Each single test can result in the following ways:

– If a given access attempt should be permitted, then obviously a successful access is a test pass. A `SecurityException` indicates a test failure.
– If a given access attempt should not be permitted, then a `SecurityException` (or some other exception specified for a given access rule, e.g. `SystemException`) indicates a test pass. A successful access attempt clearly indicates a test failure.
– Each test can result in an exception other than the one mentioned (or lack thereof) by the specification. This again indicates a test failure.

4 Test Results

In this section we describe test results for tests that we either think are interesting for some reason or that exhibit some card's non-compliance. None of the failed tests seem to open serious security holes in a Java Card implementation. Some cards are not fully compliant to the specification, but typically in ways that are harmless, in the sense that they do not introduce security risks, and even very unlikely to cause problems with compatibility or portability. Only one case of non-compliance that we found – card `A_221` ignoring the restriction on access via a shareable interface when the applet is active on another logical channel, as described in Section 4.6 – could be consider a security problem, in that the card is more permissive than the specifications allow.

4.1 "Direct" Operation on Shareable Objects

The Java Card specifications implicitly assume that methods of shareable interfaces are invoked on shareable object acquired through the shareable interface mechanism of the Java Card API, in the following way:

```
AID a = JCSystem.lookupAID(servAID, (short)0, (byte)servAID.length);
SInterface sio =
  (SInterface)JCSystem.getAppletShareableInterfaceObject(a, (byte)0);
```

where SInterface is an interface type that extends Shareable. This way the static type of the acquired object is always some shareable interface. Then, because of static type limitations, it is only possible to invoke methods of that interface (i.e. with the invokeinterface opcode). Moreover, the object in question can be only cast to shareable interfaces it implements, and not to any class types.

It is, however, possible to acquire a shareable object with a static type different from the defined shareable interface. This is possible to achieve by getting the object directly from another context through a public static method, following this scenario:

```
public class SomeClass implements SInterface {
  void method1() { // declared in SInterface, which extends Shareable
    ...
  }
}

public class ServerApplet extends Applet {
   // sc belongs to ServerApplet, but is Shareable
   SomeClass sc = new SomeClass();

   // normal way of giving away the shareable interface
   public Shareable getShareableInterfaceObject(AID clAID, byte param) {
     return sc;
   }

   // "static" way of giving away the shareable interface
   public static SomeClass getSomeClass() {
     return sc;
   }
}

public class ClientApplet extends Applet {
   ...
   SomeClass o = ServerApplet.getSomeClass();
   o.method1(); // Should this be possible? Specification says no...
   ...
}
```

(We made one presentation shortcut here. Technically the getSomeClass method cannot be placed in the ServerApplet package, but it is possible to achieve the same effect with this method residing in a separate library package.) In this case the static type of the object can be any type between Object and the actual runtime (dynamic) type of the object, exactly as in the example. (Here the dynamic type of o is SomeClass.)

The question now is whether it should still be possible to access shareable methods of this object o. According to the rules defined in the specification it is forbidden [11, Section 6.2.8.4] – the static type of the reference that a method is invoked on (here o) statically has to be of a shareable interface type [11, Section 6.2.8.6]. On the other hand, if we know that we deal with a shareable object and we implicitly know the shareable methods, it might be possible to call those methods directly "on the whole" (so to say) object o, i.e. the o.method1(); call could be legal.

All the cards required the static type of a shareable object to be the shareable interface before any methods could be called.

4.2 Querying the Shareable Status

According to the specification it is not possible to check if an object is an instance of the Shareable interface if that object belongs to a different context

[11, Section 6.2.8.10].[3] All cards but one (`A_221`) indeed prohibit such a check – a `SecurityException` is thrown. `A_221` simply returns `false` instead. Note that both behaviours give the same overall result: we can find out if the object is shareable or not, just the result is reported in a different way. However, strictly speaking, the second behaviour is not compliant with the specification.

4.3 Privileged API Methods

The purpose of this test was to find out whether it is possible to convince privileged API methods to read data outside of the context of the client calling such a privileged method. Although it was not possible to break the security this way the test revealed some slight differences in the implementation of some API methods (notably `equals` methods of the `AID` class). Let us look at the first method:

```
public boolean equals(Object o);
```

The checks that this method has to do are the following:

– *firewall check:* if the caller of the method does not have the right to access `o`, i.e. the caller context is different from the context of `o`, then a `SecurityException` should be thrown;
– *instance check:* if object `o` is not of type `AID` then `false` shall be returned;
– AID comparison: if the firewall allows access to `o` and `o` is of type `AID` then the actual comparison of the AID bytes takes place.

Performing the first two checks in a different order can produce a different result. We have indeed observed these different behaviours on different cards: for a non-AID object belonging to another context passed to the `equals` method, some cards (`A_221`, `A_211`, `B_22`) give a `SecurityException`, while other cards (`B_211`, `B_221`) return `false`. We consider the second behaviour non-compliant, because it lets an applet check if an object that belongs to another context is an `AID` object or not (but nothing more). Again, although this seems harmless, this is not compliant to the specification. The documentation of the `equals` method says that the method will throw a `SecurityException` "if `o` object is not accessible in the caller's context". Although this by itself does not require the firewall check to be the first one, another part of the specification forbids applets to check instances of objects belonging to other contexts [11, Section 6.2.8.8].

The implementation of the second `equals` method in the `AID` class:

```
public boolean equals(byte[] bArray, short offset, byte length);
```

also shows some implementation differences between the cards, although here truly harmless. The two checks that can be made in an arbitrary order are (i) the firewall check for the byte array `bArray`, (ii) whether the `length` parameter is equal to the AID length stored in the object. `A_211` checks the length first, all the other cards do the firewall check first. Note that doing the second check first does not reveal any information about the byte array `bArray`, but it can reveal information about the length of the AID (which is a byte array) being queried. However, this can be legally checked anyhow with the `getBytes` method.

4.4 Reported Exceptions on Array Access outside of the Context

Practically all firewall violations should manifest themselves with a `Security-Exception`, with the notable exception of a specific scenario described in the next

[3] This in itself seems to be circular logic.

section, when a clear-on-deselect array is created, and a `SystemException` should be reported. One of the cards (`A_211`), however, reports a wrong exception on attempt to access an array belonging to another context. Instead of a `SecurityException` a `SystemException` is reported. This again is a violation of the specification, but a safe one, in that it does not introduce any security risk.

4.5 Clear-on-Deselect Arrays

Transient arrays that are cleared on applet deselection are required to behave in a certain way: access to (or creation of) such arrays is only allowed if the current context is the context of the currently selected applet. In particular it means that an object (applet) cannot access its own clear-on-deselect array when the object is not running in the context of the currently selected applet. (The motivation for this restriction is that the contents of such arrays should be regarded sensitive, as this is the reason to making them clear-on-deselect.) This situation occurs when the applet's method is invoked from the primary context after a context switch through a shareable interface. This rule also applies even if the attempting context is selected on another logical channel.

All the cards behave accordingly to these rules. However, one card (`A_221`) extends this rule to clear-on-reset arrays when a creation of an array is attempted. That is, clear-on-deselect arrays cannot be accessed or created (as stipulated by the specification), while clear-on-reset arrays can be accessed (correct behaviour), but cannot be created (incorrect behaviour). Again, this violation is safe (in fact one could argue that it introduces more security), nevertheless limits the functionality of the card.

4.6 Multiselectable Applets and Specification Redundancy

The newer Java Card specification (2.2 onwards) can make use of the mechanism of logical channels. More than one applet can be selected at the same time on different logical channels. This applies to applets within the same package/context, but also to applets in different packages/contexts. Because of this possibility some addition firewall restrictions are included in the specification. Notably, access to a shareable interface is forbidden if the context of the object being accessed (the server object) is active on another logical channel [11, Section 6.2.8.6]. One of the 2.2.x cards (`A_221`) ignores this rule and grants the access regardless of whether the server applet is active on another logical channel or not. Note that this is potentially dangerous, in that it could lead to a security problem for a particular applet, even though this does not seem very likely. All other 2.2.x cards (`B_22`, `B_221`) respect this rule.

Then, note that if this rule is properly implemented, then one of the comments [11, Section 6.1.5] in the Java Card specification becomes redundant. The comment states that the clear-on-deselect array access rules we discussed in Section 4.5 "also apply even if the attempting context is selected on another logical channel". That is, the rule is extended to multiselectable setting. The only scenario (that we can think of) to violate the rule from Section 4.5 is when the method that accesses a clear-on-deselect array is called through a shareable interface. Then, because of the context switch, the current context is not the same as the context of the currently selected applet. In the multiselectable scenario the shareable interface method call is forbidden in the first place, thus the clear-on-deselect array access in the multiselectable scenario should never happen. Thus, the comment mentioned above seems obsolete. The only other way to violate the rule is when JCRE accesses the array. For JCRE, however, the access is always granted following the statement in [11, Section 6.2.3]. Although this issue can be seen as a simple specification redundancy (which in some cases is good) it also can lead to confusion, which in turn may lead to faulty implementations.

Finally, the newest Java Card specification (2.2.2) also enforces a similar rule [11, Section 6.2.7.2] when the acquiring (apart from accessing) of a shareable interface is attempted – getting a shareable object is forbidden if the server applet is selected on another logical channel. None of the cards we tested enforce this rule, but they do not have to as none of the cards implement the 2.2.2 specification. It would be very interesting to see if 2.2.2 cards[4] enforce this rule.

4.7 Stack Overflow

One of our tests checks to see whether the current context is not corrupted by a stack overflow (the idea for this particular test was borrowed from JCWorkBench tests). This was tested by causing an infinite recursive method call. All but one card behaved properly by throwing an error and restoring the initial context. The one card (`B_22`) seems to have strange control over stack overflows: upon stack overflow the card terminates with the status word `6F0C` and there is no error or exception that can be caught. In this case it is not possible to actually establish whether context information is maintained properly, but it is clearly no longer an issue as execution is halted.

4.8 Other

Finally, there is a publicly available emulator for `A_221`. This emulator behaves *exactly* the same way during the firewall test as the actual card.[5] This suggest that it is a true emulator, one that actually runs the same API implementation as the actual card. (Then the interesting question is whether the API implementation could be extracted/decompiled from the emulator.)

5 Conclusions

Our testing of the Java Card firewall revealed only minor issues where implementations did not comply with the specification. Most of them were 'safe' violations of the specs, in that they would not introduce any security risks. They might have a negative impact on interoperability/portability, although we do not think this is very likely to occur in practice for the issues we found.

The only 'unsafe' violation of the specifications we found was card `A_221` ignoring the restriction on access via a shareable interface when the applet is active on another logical channel (Section 4.6). This does not immediately result in security problems, but it could lead to security problems in particular applications that use shareable interface.

All this suggests that test suites used in development are incomplete, and also that the Technology Compatibility Kit (TCK) for Java Card, the official test suite to determine compliance to the standard, can be improved. It is a pity that the TCK is not public (it is only available to Java Card licencees), as it means we cannot see how we could contribute to its improvement.

It is also clear that there are places in the specification that allow different interpretations (e.g. in which order the checks in the `AID.equals` method should be

[4] We are aware of one such card, but it was not possible to get access to it.

[5] The behaviour is so accurate that even direct reading of reference values, as described in [9], give similar results.

done). It is, however, a bit disturbing that a seemingly straightforward specification is not really implemented correctly by many of the cards. Although the problems we discovered are harmless, they question Java Card platform interoperability. This in turn puts a question on the sense of developing a formal model of the firewall. If the formal model based on the official specification does not reflect the actual card behaviour, then it cannot be used for sound proofs of Java Card program security w.r.t. firewall properties in a real-life setting.

Acknowledgements

This work and our project (`http://www.win.tue.nl/pinpasjc/`) is supported by the research program Sentinels (`http://www.sentinels.nl`). Sentinels is financed by the Technology Foundation STW, the Netherlands Organisation for Scientific Research (NWO), and the Dutch Ministry of Economic Affairs. We would also like to thank Marc Witteman for interesting personal discussions and providing us with the JCWorkBench tool.

References

1. June Andronick, Boutheina Chetali, and Olivier Ly. Using Coq to verify Java Card applet isolation properties. In *Proceedings of 16th Theorem Proving in Higher Order Logic, Roma (TPHOL'2003)*, volume 2758 of *LNCS*, pages 335–351. Springer, 2003.
2. Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer-Verlag, 2007.
3. Werner Dietl, Peter Müller, and Arnd Poetzsch-Heffter. A type system for checking applet isolation in Java Card. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *LNCS*, pages 129–150. Springer, 2004.
4. Engelbert Hubbers, Wojciech Mostowski, and Erik Poll. Tearing Java Cards. In *Proceedings, e-Smart 2006, Sophia-Antipolis, France, September 20–22*, 2006.
5. Engelbert Hubbers and Erik Poll. Transactions and non-atomic API calls in Java Card: specification ambiguity and strange implementation behaviours. Dept. of Computer Science NIII-R0438, Radboud University Nijmegen, 2004.
6. Michael Montgomery and Ksheerabdhi Krishna. Secure object sharing in Java Card. In *Proceedings of the USENIX Workshop on Smartcard Technology (Smartcard '1999), Chicago, Illinois, USA, May 10–11*, 1999.
7. Wojciech Mostowski. Formal reasoning about non-atomic Java Card methods in Dynamic Logic. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *Proceedings, Formal Methods (FM) 2006, Hamilton, Ontario, Canada*, volume 4085 of *LNCS*, pages 444–459. Springer, August 2006.
8. Wojciech Mostowski. Fully verified Java Card API reference implementation. In Bernhard Beckert, editor, *VERIFY'07 4th International Verification Workshop*, volume 259 of *CEUR WS*, July 2007.
9. Wojciech Mostowski and Erik Poll. Malicious code on Java Card smartcards: Attacks and countermeasures. To be submitted.
10. Daniel Perovich. Secure object sharing development kit for Java Card. In *Proceedings, VerifiCard Annual Meeting*, 2002. Available at `http://www-sop.inria.fr/lemme/verificard/2002/programme.html`.
11. Sun Microsystems, Inc., `http://www.sun.com`. *Java Card 2.2.2 Runtime Environment Specification*, March 2006.