Malicious Code on Java Card Smartcards: Attacks and Countermeasures

Wojciech Mostowski and Erik Poll
Digital Security
Radboud University Nijmegen

To be presented at CARDIS‘2008
Overview

- Background and motivation

- Ways to *create* type confusion
  - experiments on actual cards

- Ways to *exploit* type confusion
  - experiments on actual cards
  - runtime countermeasures used

- Conclusions
Background

- Java Card smartcard allow **multiple applets** to be installed
  - installation strictly **controlled by digital signatures**
  - **or completely disabled**
    - eg on Dutch Java Card e-passport

- Most JavaCard smartcards have **no bytecode verifier**
  - could malicious, ill-typed applets do any damage?
  - not just to other applets, but also to platform
    - eg retrieving bytecode of platform implementation

- Java Cards do have a **firewall**
  - can this compensate for absence of bvc?
Two lines of defence on Java Card platform

- **Type safety**
  - enforced by bytecode verifier at *installation time*
  - optional; most cards use code signing instead

- **Firewall**
  - enforced by VM at *runtime*
  - restricts interactions between applets that type system allows
  - quite tricky!!

Are these defences *complementary* or *defense-in-depth*?
- what guarantees can firewall make about ill-typed code?
Java security: type-safety + visibility

Java platform (JRE = VM + API)
Java security: type-safety + visibility + sandbox

<table>
<thead>
<tr>
<th>untrusted applet</th>
<th>public access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>private access</td>
</tr>
<tr>
<td></td>
<td>disallowed API call</td>
</tr>
<tr>
<td></td>
<td>STOP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>trusted applet</th>
<th>allowed API call</th>
</tr>
</thead>
<tbody>
<tr>
<td>STOP</td>
<td></td>
</tr>
</tbody>
</table>

sandbox

Java platform (JRE = VM + API)
Java security: type-safety + visibility + sandbox

Java platform (JRE = VM + API)

untrusted applet

trusted applet

sandbox

no protection if trusted applet or API exposes
- public field
- exposes too much functionality
- leaks reference

public access

private access

disallowed API call

allowed API call

public field

exposes too much functionality

leaks reference
JavaCard security: ... + firewall

- runtime checks to prevent
  - exposing public field
  - exposing too much functionality
  - leaking reference

Java platform (JRE = VM + API)

untrusted applet

trusted applet

JCRE entry points

STOP

ok
Ill-typed code on Java Card

- NB Java Card specifications only define behaviour of well-typed programs
  - For ill-typed code, *all* bets are off....
    - This is case for VM spec, API specs, and JCRE specs
    - Eg a card could do a complete memory dump if a type error occurs. The specs allow this, but it's clearly unwanted.

- Only way to find out what happens:
  - test some cards
Rest of this talk

- Ways to *create* type confusion
  - how can be trick the VM in accessing *the same piece of physical memory* via references with *different (incompatible) types*?

- Ways to *exploit* type confusion to do some damage
  - *'illegally' read or write memory* in ways that should not be allowed
Way to create type confusion

• byte code editing
  • edit bytecode by hand to introduce type errors
    • or use some tool, eg by ST Microelectronics

• abusing shareable interface mechanism
  • two well-typed applets with type mismatch in shareable interface between them

• abusing transaction mechanism
  • exploring bug in transaction mechanism implementation

• fault injections?
  • introduce hardware fault (eg by laser) to corrupt memory that stores bytecode
Creating type errors with shareable interface

Both applets type-correct (individually), compilable, and loadable.
Creating type errors using transactions

```java
class MyApplet extends Applet {
    short[] s; // instance field
    byte[] b; // instance field

    void someMethod() {
        short[] local = null;
        JCSystem.beginTransaction();
        s = new short[1]; s[0] = 24;
        JCSystem.endTransaction();
        ...}"
```

- `s` is either allocated *and* initialised, or neither, even if execution is interrupted by a card tear
- `s` reset to null if a card tear occurs during transaction
Creating type errors using transactions

```java
class MyApplet extends Applet {
    short[] s;  // instance field
    byte[] b;   // instance field

    void someMethod()
    {
        short[] local = null;
        JCSystem.beginTransaction();
        s = new short[1]; s[0] = 24;
        local = s;
        JCSystem.abortTransaction(); // resets s to null
        b = new byte[10];
        if ((Object)b == (Object)local)...// true on some cards!!!
    }
}
```

*buggy transaction mechanism reset only s to null, not local*
One role of formal methods

(Too) hard to formalise

⇒

Hard to implement

⇒

Security problems are not unlikely....

For example, the transaction mechanism is very tricky when allocating objects inside transactions

- see Nicolas Rousset's thesis, Chapter 3
### Experiments creating type confusion

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>on-card</td>
<td>11</td>
<td>21</td>
<td>11</td>
<td>2</td>
<td>21</td>
<td>11</td>
<td>1'</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>bcv? bytetcode</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>yes</td>
<td>yes</td>
<td>✓</td>
</tr>
<tr>
<td>abusing editing</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>abusing shareable</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>transact</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

*ill-typed code possible on card C211 *despite* bcv!!

- because of buggy transaction mechanism
- cards with bcv don't allow shareable interfaces
- and hence are not standard-compliant?
Ways to exploit with type confusion

- confusing byte arrays and short arrays
  - possibly accessing twice as much memory
- accessing array as object
  - possibly set the length field
- accessing object as array
  - possibly doing pointer arithmetic (using numeric value as references)
- confusing objects of various types
  - possibly accessing outside memory or doing pointer arithmetic
What if VM can be tricked to access object \texttt{A} \texttt{a} as if it is of type \texttt{B}?

We might be able to

- access memory outside bounds \textit{(namely a.z)}
- do pointer arithmetic \textit{(using a.y)}
- modify final fields \textit{(namely a.x)}
Accessing byte array as short array

```java
byte[] b = { 23, 24}; // b.length = 2
```

If we access `byte[] b` as `short[] s`, then

- what is `s.length`?
- what is `s[1]`?

If VM can be tricked in treating `byte[]` as `short[]`, physical array size might double,

- allowing access outside array bounds
Accessing object as array (1) [M Witteman, RSA2003]

```java
public class FakeArray {
    short length = 0x7FFF;
    short x = 23;
}
```

If VM can be tricked in treating `FakeArray` as `short[]`, maybe array lengths can be set

- accessing **memory way outside the object's bounds**
- depending on layout of objects and arrays in memory
Accessing object as array (2)

```java
public class MyObject {
    A a = new A();
    B b = new B();
    C c = new C();
}
```

Treating `MyObject o` as a `short[] s`, what happens with

- `s[0] = s[1];`  
  - swapping references like this works on some cards

- `s[0] = 24612;`  
  - spoofing a reference like this fails on nearly all cards
Runtime defense mechanisms

Some cards employ runtime countermeasures:

- **Physical Bounds Checking (PBC)**
  array bounds are checked using physical sizes rather than logical sizes
  - confusing `byte[]` and `short[]` becomes harmless

- **Object Bounds Checking (OBC)**
  object bounds checked at runtime just like array bounds
  - confusing objects and arrays becomes less harmfull;
    - no access beyond object's original size

- **Runtime Type Checking (RTC)**
  object types are checked at runtime for every VM step
  - all attempts at type confusion become harmless
### Experiments running ill-typed code

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>byte-short[ ]</td>
<td>11</td>
<td>21</td>
<td>11</td>
<td>2</td>
<td>21</td>
<td>11</td>
<td>bev</td>
<td>11</td>
</tr>
<tr>
<td>object as array</td>
<td>PB</td>
<td>O</td>
<td>R</td>
<td>O</td>
<td>R</td>
<td>bc</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>array as object</td>
<td>C</td>
<td>B</td>
<td>T</td>
<td>B</td>
<td>T</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>reference switching</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>nt</td>
</tr>
<tr>
<td>~ in AIDs</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>nt</td>
</tr>
<tr>
<td>reference spoofing</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Reference switching in AID objects

package javacard.framework
public class AID {
    final byte[] theAID;
    ...
}

- reference switching on some cards allows theAID field in AIDs (Applet IDentifiers) to be changed to point to other byte arrays
  - this allows system-owned AIDs to be changed
  - AIDs are used for identifying applets on the card...
Conclusions

- Many attacks, some with harmful results
- On-card bcv not sufficient
  - if there are bugs in transaction mechanism...
  - Also, on-card bcv limits functionality:
    - no Shareable Interfaces between applets
- (Increasingly?) cards employ runtime countermeasures
  - runtime checks more robust than static checks!
  - runtime typechecking is best countermeasure
    - downside: performance overhead?
- All this applies only to open cards
  - no threat on most (all?) Java Cards in the field