# Memory Forensics of a Java Card Dump

jean-louis.lanet@inria.fr

Cardis 2014 Paris
Nov. 5-7 2014

# Episode 2

- Previous episode: how to obtain a dump
- Hypothesis
- Find the code
- Reverse it
- Conclusion

# Memory Dump


- At that time we have a binary file representing the memory,
- Reversing is a hard task,
  - E2prom has no region,
  - Several heaps,
  - Several binary languages,
  - Unknown bye codes,
  - Sometime masked sometime encrypted.
- Task prone to error and no tool to automatically reverse it,
- The objective: obtain from the binary dump the Java source file.

$\underbrace{1111}_{F}\,\underbrace{0100}_{4}$

# From binary to source

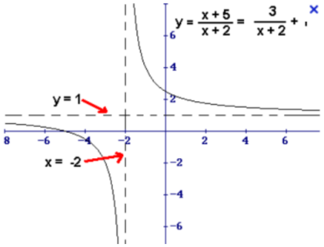
- Starting point is the dump file where somewhere is the method area,
- Reversing process
  - Isolate the method area,
  - Regenerate a CAP file,
  - Tokenize the CAP

  - Use the CAP2Class tool
  - Use a Class2Java tool

# Memory Carving

- Regenerate the memory regions
  - Extract the Java Byte code area from the rest,
  - Remaining could be:
    - System Data, Application Data, VM Data, Native code
- Usual approach brute force
  - Verify a legal control flow graph,
  - Adapted to small pieces of code,
  - We can not use byte code interpretation due to illegal byte code,
  - We need a heuristic approach.

# Limit of the approach




- It does not work if:
  - the dump refers to encrypted byte code area not obtained with the VM but using an array extension,
  - the encrypted code has different key for different security context if obtained by the VM using a **`getstatic`**,
  - the card use a dynamic **`xor`** (Razandralambo, 2012)
- Works well:
  - Code is in plain text
  - Use a static **`xor`**.

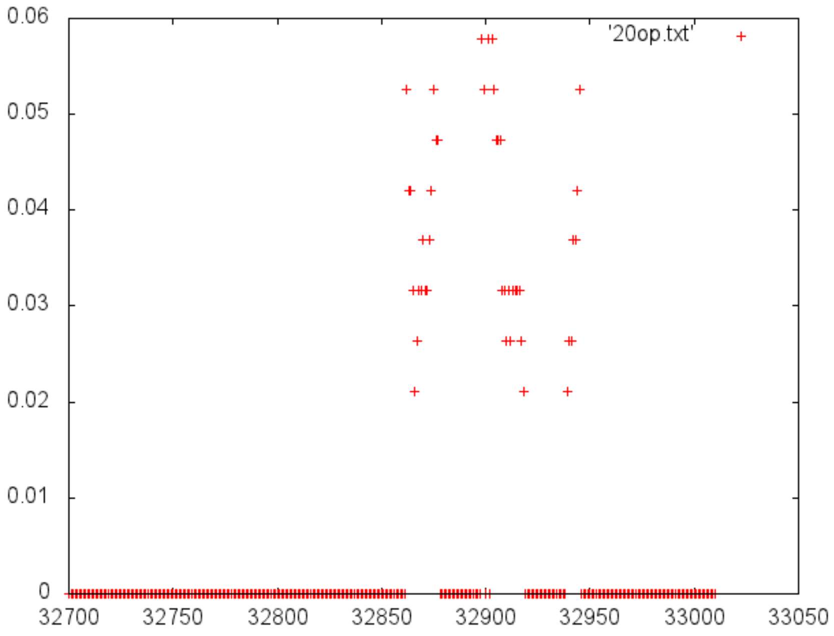
# Memory Carving

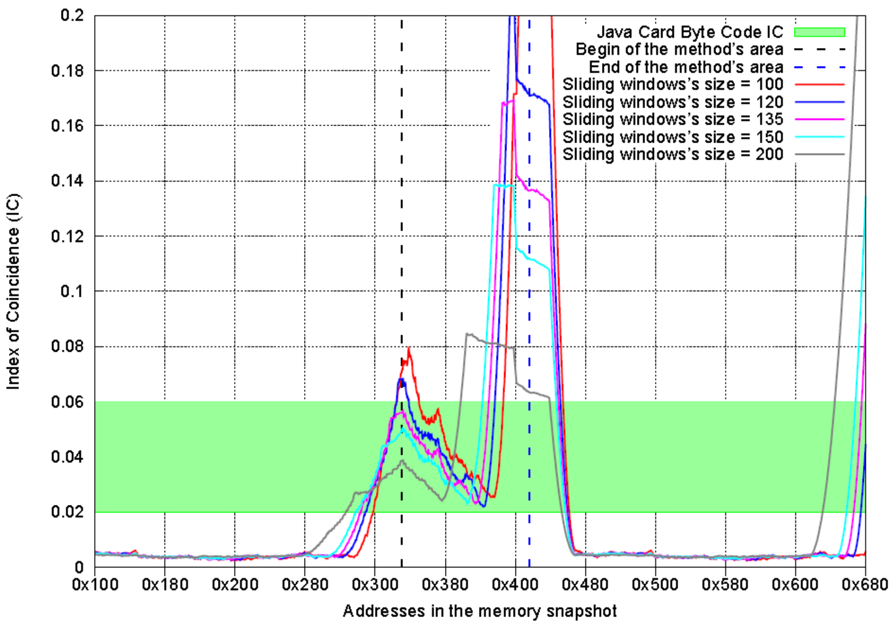

- Forensic Memory Carving,
  - Using language recognition,
  - Java and Assembly area,
  - Array  and Object structure
- Index of coincidence

$$\mathrm{IC} = \frac{\displaystyle\sum_{i=1}^{c} n_i(n_i - 1)}{N(N-1)/c}$$

  - The value of IC for Java Card byte code in a CAP file is between 0.02 and 0.06

# Memory Carving



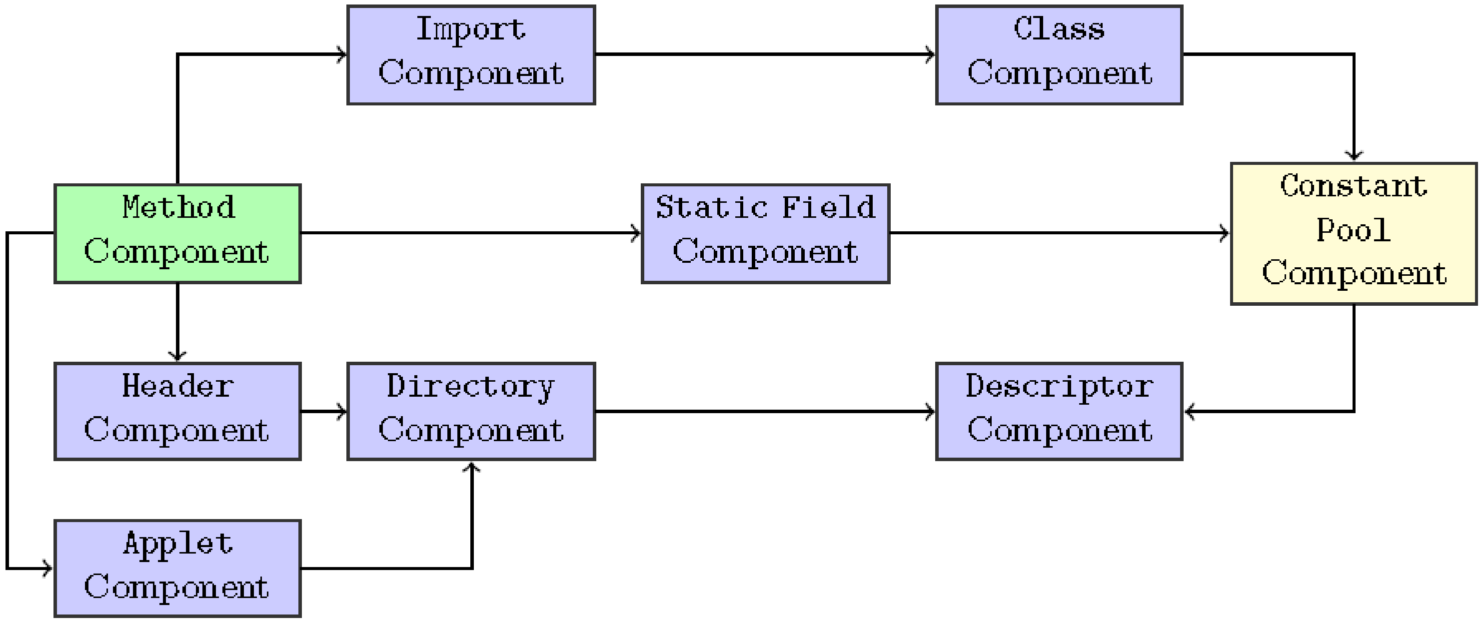



# Symbolic execution

- Building the different CFG,
- By hypothesis we do not have the **`*.exp`** file of the applet,
- Identifying the beginning of each method,
  - Checking the stack evolution in term of type system,
  - Isolate the unknown instructions with their effects on the stack,
- As a result a set of grouped methods with 2..4 entry points:
  - **`process`**, **`install`**, **`select`**, **`deselect`**,
  - The others are private methods plus the constructor,
    - (**`aload_0`**; **`invokespecial 0`**;…)
    - Sometime proprietary instructions…

# Reversing

- At that step we have identified the different method areas,
- We have to rebuilt the CAP components from the method component.



# Resolve the names and rebuild

- Thanks to (Hamadouche, 2012) we have the relationship between addresses and method names,
  - This is the way to identify **`register()`**, **`ifSelectingApplet()`** that characterize **`install()`** and **`process()`**,
  - It allows to define the **`import`** component and then the **`class`** component,
- Rebuild the **`header`** and the **`applet`**,
- Issue:
  - the **`staticField`** component initialization: current value or default value
  - the accessor of the attributes defined in the **`class`** are lost.

# Finish the CAP

- Some instructions in **`method`** require parameters that must be un resolve,
- Generate the tokens and build the **`reference location`** and the **`constant pool`** components.
- Build the **`descriptor`** component that has all the offsets of each component.

# Obtain the source code

- Students designed a “Partial Linked Cap to Unresolved Cap” tool,
- Validated using the BCV,
- Not completely automated,
- But no reason to not succeed,
- When packaged could be open source.

```
// method component
040062 0019 007e 0009 0062 8019 00
85 0000 0083 8007 007d 0000 0092 8002
00
75 0000 01 10 18 8d08 9718 8b01 017a
0230
8fff ac3d ccff ee3b 7a01 1004 7800 107a
0010 7a00 107a 0140 0478 0110 1042 7801
1004 7800 107a 0323 198b 0101 2d18 8b01
0360 037a 1006 8118 1a10 0781 181c 1008
8118 1c1a 0310 ca38 1a04 10fe 3811 9999
8d08 c670 1c2e 1167 898d 08c6 1199 998d
08c6 700d 2804 1199 998d 08c6 1504 937a
```

```
.method {
  handler_count : 4
  exception_handler[0]{
      start_offset      : 98
      bitfield          : 25
      handler_offset    : 131
      catch_type_index  : 4
      }
  exception_handler[1]{
      start_offset      : 98
      bitfield          : -32743
      handler_offset    : 146
      catch_type_index  : 0
      }
  exception_handler[2]{
      start_offset      : 131
      bitfield          : -32761
      handler_offset    : 146
      catch_type_index  : 0
      }
  exception_handler[3]{
      start_offset      : 146
      bitfield          : -32766
      handler_offset    : 146
      catch_type_index  : 0
      }
method_info[0] // @0021= {
      // flags      : 0
      // max_stack  : 1
      // nargs      : 1
      // max_locals: 0
Bytecodes...
      }
... n method_info[]
}
```

# Conclusion

- This engineering work has been done by students of a master degree (M1) from the University of Limoges during their Java course,
- It was a 60 hours development project (5 students), around 300 hours,
- Entirely written in Java, could be provided as an open source project if they want to package their work,
- A good introduction to Java Card course.

# Question ?