Security Vulnerability Notice

SE-2019-01-ORACLE

[Security vulnerabilities in Java Card, Issues 1-18]

DISCLAIMER

INFORMATION PROVIDED IN THIS DOCUMENT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW NEITHER SECURITY EXPLORATIONS, ITS LICENSORS OR AFFILIATES, NOR THE COPYRIGHT HOLDERS MAKE ANY REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE OR THAT THE INFORMATION WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS, OR OTHER RIGHTS. THERE IS NO WARRANTY BY SECURITY EXPLORATIONS OR BY ANY OTHER PARTY THAT THE INFORMATION CONTAINED IN THE THIS DOCUMENT WILL MEET YOUR REQUIREMENTS OR THAT IT WILL BE ERROR-FREE.

YOU ASSUME ALL RESPONSIBILITY AND RISK FOR THE SELECTION AND USE OF THE INFORMATION TO ACHIEVE YOUR INTENDED RESULTS AND FOR THE INSTALLATION, USE, AND RESULTS OBTAINED FROM IT.

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL SECURITY EXPLORATIONS, ITS EMPLOYEES OR LICENSORS OR AFFILIATES BE LIABLE FOR ANY LOST PROFITS, REVENUE, SALES, DATA, OR COSTS OF PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES, PROPERTY DAMAGE, PERSONAL INJURY, INTERRUPTION OF BUSINESS, LOSS OF BUSINESS INFORMATION, OR FOR ANY SPECIAL, DIRECT, INDIRECT, INCIDENTAL, ECONOMIC, COVER, PUNITIVE, SPECIAL, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND WHETHER ARISING UNDER CONTRACT, TORT, NEGLIGENCE, OR OTHER THEORY OF LIABILITY ARISING OUT OF THE USE OF OR INABILITY TO USE THE INFORMATION CONTAINED IN THIS DOCUMENT, EVEN IF SECURITY EXPLORATIONS OR ITS LICENSORS OR AFFILIATES ARE ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS.

Security Explorations discovered multiple security vulnerabilities in Java Card [1] technology used in financial, government, transportation and telecommunication sectors among others. A table below, presents their technical summary:

ISSUE #	TECHNICAL DETAILS	
1	origin	`baload` bytecode instruction implementation
	cause	insufficient type check
	impact	compromise of memory safety / arbitrary read access to card memory
	status	verified
2	origin	`bastore` bytecode instruction implementation
	cause	insufficient type check
	impact	compromise of memory safety / arbitrary write access to card memory
	status	verified
3	origin	`javacard.framework.Util` class
	cause	insufficient type check in `arrayCopy` method implementation (*src* argument)
	impact	compromise of memory safety / arbitrary read access to card memory
	status	verified
4	origin	`javacard.framework.Util` class
	cause	insufficient type check in `arrayCopy` method implementation (*dst* argument)
	impact	compromise of memory safety / arbitrary write access to card memory
	status	verified
5	origin	`javacard.framework.Util` class
	cause	insufficient type check in `arrayCopyNonAtomic` method implementation (*src* argument)
	impact	compromise of memory safety / arbitrary read access to card memory
	status	verified
6	origin	`javacard.framework.Util` class
	cause	insufficient type check in `arrayCopyNonAtomic` method implementation (*dst* argument)
	impact	compromise of memory safety / arbitrary write access to card memory
	status	verified
7	origin	CAP file handing / verification
	cause	unchecked value of `internal_ref` offset of `CONSTANT_StaticFieldref_info` structure
	impact	compromise of memory safety / arbitrary read and write access to card memory
	status	verified
8	origin	CAP file handing / verification
	cause	`COMPONENT_ReferenceLocation` content inconsistent with Constant Pool
	impact	incomplete linking of `CONSTANT_InstanceFieldref` entries
	status	verified
9	origin	`getfield_a` bytecode instruction group implementation
	cause	unchecked value of token field
	impact	compromise of memory safety / arbitrary read access to card memory
	status	verified
10	origin	`getfield_b` bytecode instruction group implementation
	cause	unchecked value of token field
	impact	compromise of memory safety / arbitrary read access to card memory
	status	verified
11	origin	`getfield_s` bytecode instruction group implementation
	cause	unchecked value of token field
	impact	compromise of memory safety / arbitrary read access to card memory
	status	verified

12	origin	`getfield_i` bytecode instruction group implementation
	cause	unchecked value of token field
	impact	compromise of memory safety / arbitrary read access to card memory
	status	verified
13	origin	`putfield_a` bytecode instruction group implementation
	cause	unchecked value of token field
	impact	compromise of memory safety / arbitrary write access to card memory
	status	verified
14	origin	`putfield_b` bytecode instruction group implementation
	cause	unchecked value of token field
	impact	compromise of memory safety / arbitrary write access to card memory
	status	verified
15	origin	`putfield_s` bytecode instruction group implementation
	cause	unchecked value of token field
	impact	compromise of memory safety / arbitrary write access to card memory
	status	verified
16	origin	`putfield_i` bytecode instruction group implementation
	cause	unchecked value of token field
	impact	compromise of memory safety / arbitrary write access to card memory
	status	verified
17	origin	`swap_x` bytecode instruction implementation
	cause	unchecked instruction argument (N)
	impact	overwrite of JC runtime stack / potential native code execution
	status	verified
18	origin	CAP file loader / verification
	cause	unchecked flags field of `method_header_info` structure of `COMPONENT_Method`
	impact	direct invocation of inaccessibe / unexported native methods
	status	verified

Issues 1-18 were successfully verified in the environment of the most recent Oracle Java Card 3.1 SDK from Jan 2019 incorporating reference implementation of Java Card VM [2].

Gen tool

Successful exploitation of the vulnerabilities found requires generation of specially crafted CAP files (exploits). This is demonstrated by our Gen tool. This tool modifies legitimate output CAP file produced by the compiler according to target vulnerability condition to illustrate. During processing of the CAP file, the tool also processes the corresponding EXP file in order to locate methods' bytecode data by the means of type and name descriptor. As a result, the development of POC codes could be significantly facilitated (no need to seek for method's bytecode data in raw `Method` component content).

The Gen tool takes 2 arguments that correspond to the following:

- *arg0* - the index of a generation subroutine to use (target POC idx),
- *arg1* - additional argument, occasionally used by the POC code.

Table below provides description of additional argument used by the Gen tool:

ARG0	ARG1

1	*UNUSED*
2	offset used for the internal reference of `StaticFieldref`
3	token value for the `getfield_<T>` and `putfield_<T>` instructions
4	*UNUSED*
5	*UNUSED*

Vulnerability details

Below, more details are provided with respect to the discovered vulnerabilities.

Issues 1-2

Issues 1 and 2 are due to the missing type check for the object provided as a byte array argument to the `baload` and `bastore` bytecode instructions. This is illustrated upon the example of the `baload` opcode (Fig. 1).

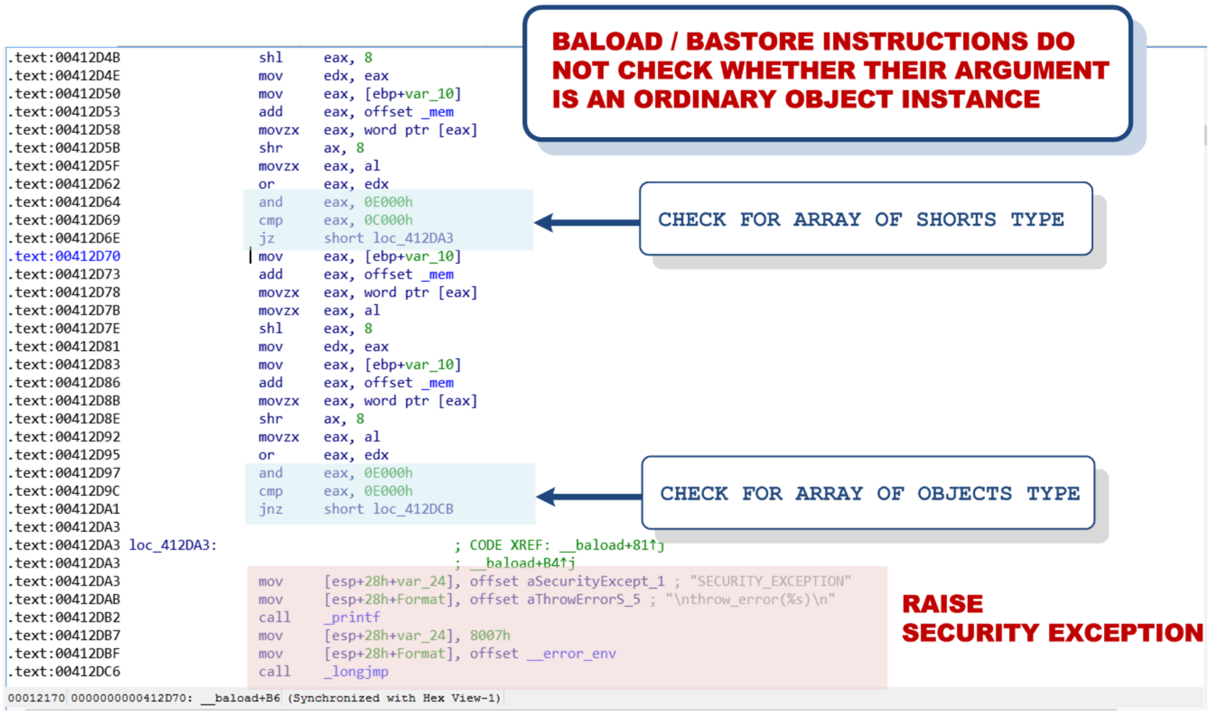

Fig. 1 Baload bytecode instruction implementation.

The code implementing `baload` instruction inspects the bits carrying information about the type of an object provided to it as an argument. It throws Security Exception if a type of an object encoded in the object header corresponds to an array of shorts (bits value `0xa000`), an array of integers (bits value `0xc000`) or an array of objects (bits value `0xe000`).

If the abovementioned checks are successfully passed, the code assumes that the object argument is an array of bytes (the only array type left). It can be of an ordinary object type though and this condition does not get detected.

As a result, ordinary Java object instances can mimic arrays of bytes (type confusion vulnerability). There is however more to this. The first instance field of such an object will perfectly match (in the context of a memory layout) the length of the array field stored in a header of a legitimate array object. As a result, object instances that are a few bytes in size can mimic byte arrays of a very large size (Fig. 2). Such objects will be treated as legitimate arrays of arbitrary user provided size by both `baload` and `bastore` instructions.

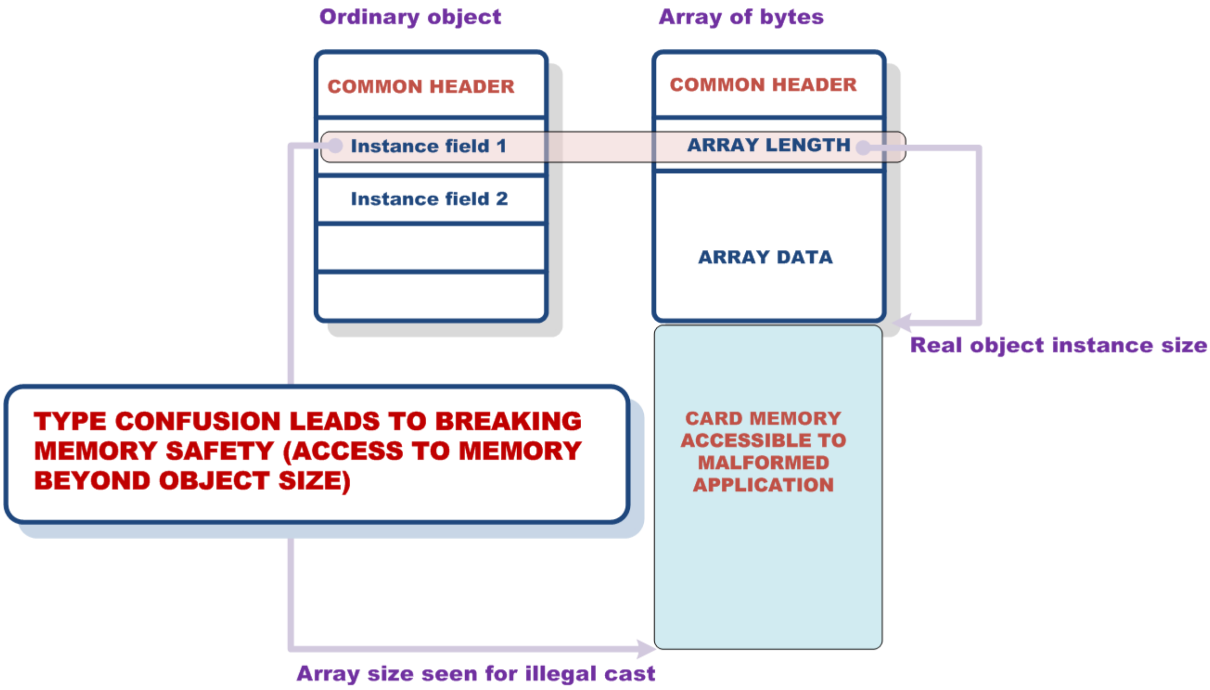

Fig. 2 Type confusion condition between ordinary object instance and array of bytes.

In our Proof of Concept code, `gen_exp1` method is responsible for generating a code sequence for an illegal type cast. The code of a target method is modified in such a way, so that instead of returning an array of bytes, it returns an object instance of a `Cast` class (change from `aload_0` to `aload_1` instruction):

```
.method public static cast2tab([BLjava/lang/Object;)[B 1 {
        .stack 1;
        .locals 0;

        .descriptor   Ljava/lang/Object;   0.0;
                L0:    aload_1      <--- CHANGE TO ALOAD_1 INSTRUCTION
                       areturn;
}
```

Issues 3-6

The implementation of various byte array copy methods is prone to similar vulnerability as described above. More specifically, neither `arrayCopy`, nor `arrayCopyNonAtomic` methods take into account the possibility to use an object instance as an input argument. This concerns both byte array arguments used as a source and destination for the array copy operation.

Again, object instances that are a few bytes in size can mimic byte arrays of a very large size. Such objects will be treated as legitimate arrays by a target array copy method.

It's worth to note that some other array copy functionality defined in Oracle Java Card environment implements proper type checks. This in particular include `arrayCopyRepack` and `arrayCopyRepackNonAtomic` methods of `javacardx.framework.util.ArrayLogic` class (Fig. 3).

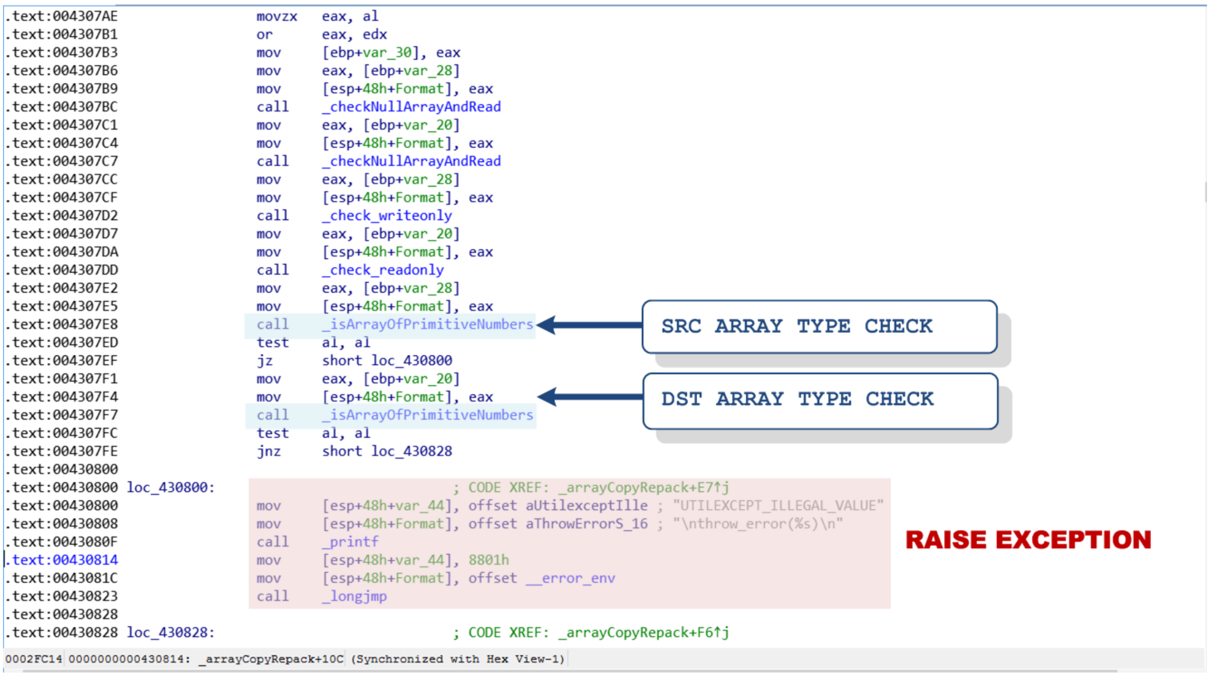

Fig. 3 ArrayCopyRepack checks of input array arguments.

In our Proof of Concept code, `gen_exp1` method is again responsible for generating a code sequence for an illegal type cast required for vulnerable `arrayCopy` and `arrayCopyNonAtomic` methods.

Issue 7

The offset item of a `CONSTANT_StaticFieldref_info` structure represents a 16-bit offset into the static field image defined by CAP file's `StaticField` component for internal static references.

This offset is not checked and can be set to arbitrary value. As a result, arbitrary accesses to memory locations beyond the static field image can be done.

The issue affects 8 bytecode instructions[1]. It is treated as a single one due to the fact that resolving of a static method reference is conducted by all of them with the use of the same single subroutine (missing security check in a code of `resolveReferenceAddress`).

In our Proof of Concept code, `gen_exp2` method is responsible for generating a code sequence making use of an overlong 16-bit offset in `CONSTANT_StaticFieldref_info`

[1] `getstatic_b`, `getstatic_s`, `getstatic_i`, `getstatic_a`, `putstatic_b`, `putstatic_s`, `putstatic_i` and `putstatic_a`.

structure referenced by a target `getstatic_s` instruction (by the means of a Constant Pool index):

```
.constantPool {
        // 0
        staticMethodRef 0.0.0()V;          // java/lang/Object.<init>()V
        // 1
        staticFieldRef short Test/dummy;    <-- CHANGE TO OVERLONG OFFSET
}
...
.method public static get_static()S 1 {
        .stack 1;
        .locals 0;

                L0:     getstatic_s 1;          <-- INDEX OF A TARGET CP ENTRY
                        sreturn;
}
```

Issue 8

CAP file's `ReferenceLocation` component contains table of offsets to bytecode locations containing indices to Constant Pool entries used by various field and method referencing instructions.

This is illustrated by the following code:

```
.constantPool {
        // 0
        instanceFieldRef short Test/dummy;
        // 1
        instanceFieldRef short Test/dummy2;
        // 2
        instanceFieldRef 0.0 Test/field_a;
        // 3
        instanceFieldRef byte Test/field_b;
        ...
 }

.method public static getfield_a()Ljava/lang/Object; 3 {
        .stack 1;
        .locals 0;
        .descriptor   Ljava/lang/Object;  0.0;

                L0:     invokestatic 7; // com/se/vulns/Test.init()V
                        getstatic_a 10; // reference com/se/vulns/Test.t
                        getfield_a 2;    // reference com/se/vulns/Test.field_a
                        areturn;
}
```

In the code above getfield_a instruction references Constant Pool entry at index 2, which contains token for the accessed instance field (field_a of Test class).

Upon CAP file loading, internal representation of this token[2] is directly stored in place of a Constant Pool index. Bytecode locations where such a "linking" should take place are

[2] for Oracle Java Card reference implementation this is simply the index of a given field instance.

contained in `ReferenceLocation` component. What's important is that prior to the described process, token value gets checked, so that it does not go beyond the size of a given object instance. In such a case, the token value simply gets trimmed.

The vulnerability is about the possibility to skip the "linking". If a target bytecode instruction referencing Constant Pool entry is omitted in the `ReferenceLocation` component, the index used as its argument will not be a subject to any modification (and checking).

In our Proof of Concept code, `gen_exp3` method is responsible for generating a CAP file illustrating Issue 8. The indices of various `getfield_<T>` and `putfield_<T>` instructions are modified to the given overlong value. The generating code also removes all references to such instructions from the `ReferenceLocation` component.

There is a potential to exploit Issue 8 for arbitrary method invocation (inaccessible / unexported to user code), but this requires a more throrough investigation of the linking mechanism conducted with respect to method invocation instructions (an internal representation of external references, whether references to unexported / native methods are valid, etc.).

Issues 9-16

The implementation of all 24 instance field access instructions[3] does not check the internal token value used as their argument. As a result, overlong token values can be provided for them and arbitrary memory content beyond target object size can be accessed.

As each triple of `getfield_<T>`, `getfield_<T>_this`, `getfield_<T>_w` instructions rely on one vulnerable routine missing token value check (`getfield_<T>_common`, Fig. 4), there are 4 vulnerabilities corresponding to `getfield` instructions (Issues 9-12). In a similar fashion, missing security checks in `putfield_<T>_common` subroutines implicates 4 additional vulnerabilities associated with `putield` instructions (Issues 13-16).

[3] `getfield_<T>`, `getfield_<T>_this`, `getfield_<T>_w`.

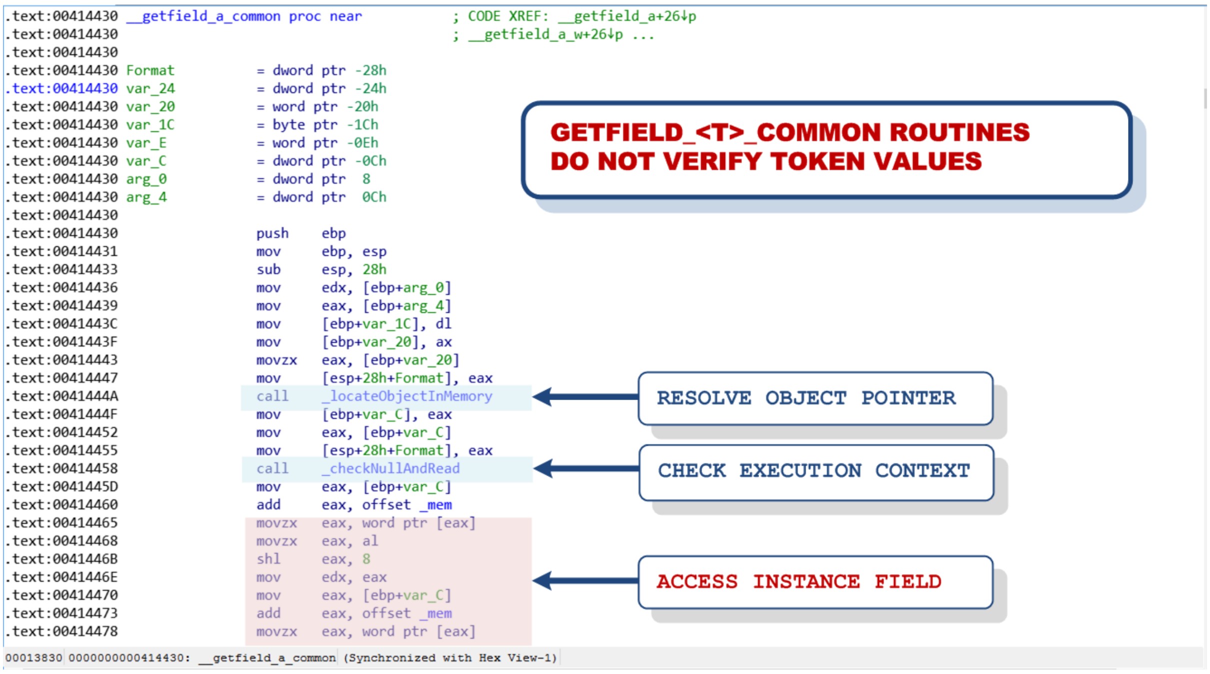

Fig. 4 The implementation of getfield_a_common.

In our Proof of Concept code, `gen_exp3` method is responsible for generating a CAP file that illustrates both Issue 8 and Issues 9-16. Issue 8 provides a means for the use of overlong token values by instance field access instructions. Issues 9-16 illustrate that such overlong token values are not checked at runtime.

It should be also possible to trigger Issues 9-16 by simply directing bytecode execution to the malicious instruction stream (through `goto` / `jsr` or conditional jump instruction).

Issue 17

Implementation of a `swap_x` bytecode instruction makes it possible to swap top *M* words on the operand stack with the *N* words immediately below (Fig. 5).

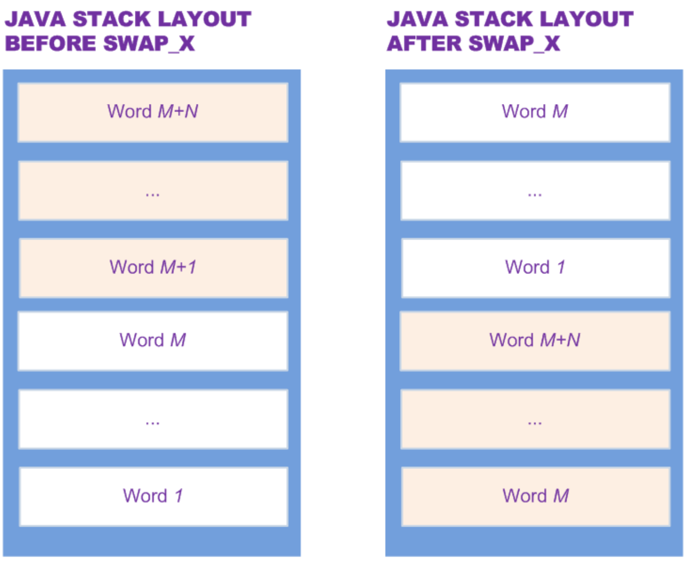

Fig. 5 Operation of swap_x instruction.

The permissible values for *N* and *M* are 1 or 2 (the instruction goal is to swap top two operand stack words). The latter value is allowed if integer types are supported in a target Java Card environment.

For larger *N*, the values pushed onto the stack can overwrite the stack frame of the invoked `swap_x` subroutine itself as stack temporary location is used for storing values popped off the Java stack. Depending on a target processor architecture, this can lead to return address / instruction pointer overwrite and native code execution (Fig. 6).

In our Proof of Concept code, `gen_exp4` method is responsible for generating a CAP file illustrating Issue 17. It generates a trigger sequence composed of multiple `push_s` bytecode instructions followed by the `swap_x` instruction. The sequence of values pushed onto Java stack needs to take into account the fact that some local variables such as loop counter, *M* and *N* themselves also get overwritten.

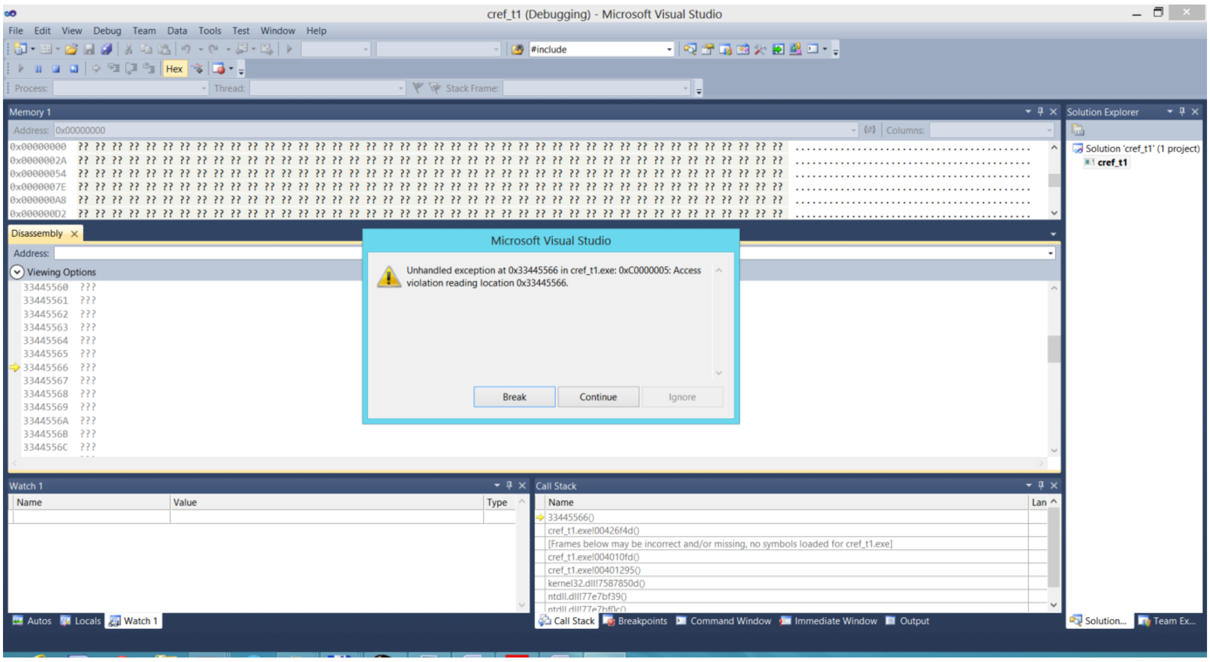

Fig. 6 Java Card process crash triggered by swap_x vulnerability.

Issue 18

A simple change of the flags field of `method_header_info` structure contained in CAP file's `Method` component to 0x02 changes target method type to native. In such a case, `nargs` and `max_locals` fields of `method_header_info` structure are not used. A one byte index is used in their place, which denote the index of a target native method to call (Fig. 7).

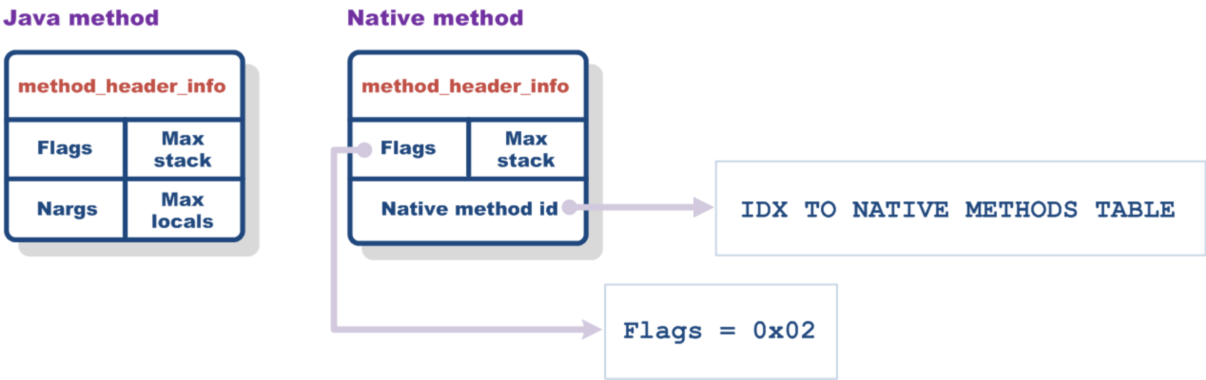

Fig. 7 Java vs. native method header.

Issue 18 can be used to call any (unexported or inaccessible to current class) native method defined in a target Java Card environment. The indices for given methods can be found in `_nativeMethods` table (Fig. 8).

```
.data:00443614                 align 10h
.data:00443620                 public _nativeMethods
.data:00443620 _nativeMethods  dd 0                    ; DATA XREF: _callnative+11↑r
.data:00443624                 dd offset _Java_NativeMethod_javacard_framework_JCSystem_isTransient
.data:00443628                 dd offset _Java_NativeMethod_javacard_framework_JCSystem_makeTransientBooleanArray
.data:0044362C                 dd offset _Java_NativeMethod_javacard_framework_JCSystem_makeTransientByteArray
.data:00443630                 dd offset _Java_NativeMethod_javacard_framework_JCSystem_makeTransientShortArray
.data:00443634                 dd offset _Java_NativeMethod_javacard_framework_JCSystem_makeTransientObjectArray
.data:00443638                 dd offset _Java_NativeMethod_javacard_framework_JCSystem_makeArrayView
.data:0044363C                 dd offset _Java_NativeMethod_javacard_framework_JCSystem_isArrayView
.data:00443640                 dd offset _Java_NativeMethod_javacard_framework_JCSystem_getAttributes
.data:00443644                 dd offset _Java_NativeMethod_javacard_framework_JCSystem_getTransactionDepth
.data:00443648                 dd offset _Java_NativeMethod_javacard_framework_JCSystem_getUnusedCommitCapacity
.data:0044364C                 dd offset _Java_NativeMethod_javacard_framework_JCSystem_getMaxCommitCapacity
.data:00443650                 dd offset _Java_NativeMethod_javacard_framework_SensitiveArrays_assertIntegrity
.data:00443654                 dd offset _Java_NativeMethod_javacard_framework_SensitiveArrays_isIntegritySensitive
.data:00443658                 dd offset _Java_NativeMethod_javacard_framework_SensitiveArrays_isIntegritySensitiveArraysSupported
.data:0044365C                 dd offset _Java_NativeMethod_javacard_framework_SensitiveArrays_makeIntegritySensitiveArray
.data:00443660                 dd offset _Java_NativeMethod_javacard_framework_SensitiveArrays_clearArray
.data:00443664                 dd offset _Java_NativeMethod_javacard_framework_Util_arrayCopy
.data:00443668                 dd offset _Java_NativeMethod_javacard_framework_Util_arrayCopyNonAtomic
.data:0044366C                 dd offset _Java_NativeMethod_javacard_framework_Util_arrayFill
.data:00443670                 dd offset _Java_NativeMethod_javacard_framework_Util_arrayFillNonAtomic
.data:00443674                 dd offset _Java_NativeMethod_javacard_framework_Util_arrayCompare
.data:00443678                 dd offset _Java_NativeMethod_javacard_framework_Util_setShort
.data:0044367C                 dd offset _Java_NativeMethod_javacard_framework_service_RMINativeMethods_copyStringIntoBuffer
.data:00443680                 dd offset _Java_NativeMethod_javacard_framework_service_RMINativeMethods_getClassNameAddress
.data:00443684                 dd offset _Java_NativeMethod_javacard_framework_service_RMINativeMethods_getRemoteMethodInfo
.data:00443688                 dd offset _Java_NativeMethod_javacard_framework_service_RMINativeMethods_getReturnType
.data:0044368C                 dd offset _Java_NativeMethod_javacard_framework_service_RMINativeMethods_deleteAllTempArrays
.data:00443690                 dd offset _Java_NativeMethod_javacard_framework_service_RMINativeMethods_isAPIException
.data:00443694                 dd offset _Java_NativeMethod_javacard_framework_service_RMINativeMethods_getAnticollisionString
.data:00443698                 dd offset _Java_NativeMethod_javacard_framework_service_RMINativeMethods_getRemoteInterfaceNumber
.data:0044369C                 dd offset _Java_NativeMethod_javacard_framework_service_RMINativeMethods_getRemoteInterfaceAddress
.data:004436A0                 dd offset _Java_NativeMethod_javacard_framework_service_RMINativeMethods_copyInterfaceNameIntoBuffer
00042028 0000000000443628: .data:00443628 (Synchronized with Hex View-1)
```

Fig. 8 Native methods table.

In our Proof of Concept code, `gen_exp5` method is responsible for generating a CAP file illustrating Issue 18. It changes the type of several dummy placeholder methods to native. When called, given unsafe methods are invoked that make it possible to read and write any address of cards' memory. These are `readByte`, `readShort`, `writeByte` and `writeShort` methods of the `com.sun.javacard.impl.NativeMethods` class.

Vulnerabilities impact

Discovered vulnerabilities make it possible to break memory safety of the underlying Java Card VM. As a result, full access to smartcard memory could be achieved, applet firewall could be broken or native code execution could be gained.

While none of the exploit codes can successfully pass off-card verification process, the vulnerabilities should be still perceived in terms of a significant weak point of Oracle Java Card VM implementation. The reasons are the following:

- the vulnerabilities could be used to compromise security of trusted chips used by financial, government and telecommunication sectors, this paves the way for their in-depth analysis, which can result in far more serious issues,
- Java Card thrives to provide secure environment for multiple applications (applets), as such no malicious application should be able to compromise security of the other one,
- split verification process is a known architectural / design weakness of Java Card, the environment should at least provide memory safety if type safety cannot be guaranteed (type safety is a direct consequence of memory safety),
- the nature of the issues undermine trust to Java Card as a secure environment and software platform eligible to run security services on smart cards and secure elements.

It should be emphasized that successful loading of a malicious applet into target card requires either knowledge of the keys or existence of some other means facilitating it (a vulnerability in card OS, installed applications, exposed interfaces, etc.). Such scenarios cannot be excluded though.

Affected versions of a reference implementation

Our Proof of Concept code were successfully tested in the environment of various versions of Oracle Java Card SDK. We verified that the following Oracle Java Card reference implementations are affected by discovered vulnerabilities:

- Java Card 3.1.0
- Java Card 3.0.5U3
- Java Card 3.0.5GA

Proof of Concept Codes usage

Each Proof of Concept code has associated `test.scr` file that defines the APDU commands illustrating vulnerabilities implemented by it. These commands are sent to the target Java Card VM instance with the use of `ApduTool` included in the Oracle Java Card SDK.

In order to test a given set of vulnerabilities, Java Card reference implementation needs to be run first:

```
c:\_SOFTWARE\Java Card Development Kit Simulator 3.1.0\bin>cref_t1.exe
Java Card 3.1.0 C Reference Implementation Simulator
32-bit Address Space implementation - with cryptography support
T=1 Extended APDU protocol (ISO 7816-3)
Copyright (c) 1998, 2019, Oracle and/or its affiliates. All rights reserved.

Memory configuration -
        Type     Base     Size     Max Addr
        RAM      0x0      0x6000   0x5fff
        ROM      0x6000   0x1efe0  0x24fdf
```

```
        E2P     0x25000 0x1ffe0 0x44fdf

        ROM Mask size =                 0x19bc2 =       105410 bytes
        Highest ROM address in mask =   0x1fbc1 =       129985 bytes
        Space available in ROM =        0x541e =                21534 bytes
Mask has now been initialized for use
```

Then, proper `run.bat` file should be executed to illustrate the operation of a given Proof of Concept code. For `baload_bastore` variant, the following output will be produced as a result of its execution:

```
c:\_WORK\PROJECTS\SE-2019-01\codes\baload_bastore>run.bat
ApduTool [v3.0.5]
    Copyright (c) 1998, 2015, Oracle and/or its affiliates. All rights reserved.



Opening connection to localhost on port 9025.
Connected.
Received ATR = 0x3b 0xf0 0x11 0x00 0xff 0x01
CLA: 00, INS: a4, P1: 04, P2: 00, Lc: 09, a0, 00, 00, 00, 62, 03, 01, 08, 01, Le
: 00, SW1: 90, SW2: 00
CAP file download section. Output suppressed.
OUTPUT OFF;
OUTPUT ON;
CLA: 80, INS: b8, P1: 00, P2: 00, Lc: 0c, 0a, a0, 00, 00, 00, 62, 03, 01, 0c, 01
, 01, 00, Le: 0a, a0, 00, 00, 00, 62, 03, 01, 0c, 01, 01, SW1: 90, SW2: 00
CLA: 00, INS: a4, P1: 04, P2: 00, Lc: 0a, a0, 00, 00, 00, 62, 03, 01, 0c, 01, 01
, Le: 00, SW1: 6e, SW2: 00
CLA: 80, INS: 10, P1: 01, P2: 02, Lc: 02, 00, 00, Le: 02, 12, 34, SW1: 90, SW2:
00
CLA: 80, INS: 11, P1: 01, P2: 02, Lc: 02, 00, 00, Le: 40, 00, 00, c0, 00, 11, 00
, 28, 1b, 00, 02, 11, 22, 33, 44, 55, 66, 77, 88, 20, 00, 11, 00, 00, 1d, 00, 90
, 00, c2, 00, 00, 20, 00, 00, 00, 23, 18, 00, c0, 00, bd, 01, 00, 20, 00, 00, 00
, 3b, 18, 00, bd, 00, 0c, 00, 00, 80, 00, 00, 00, 00, 1b, 00, 0a, a0, 00, SW1: 9
0, SW2: 00
CLA: 80, INS: 12, P1: 01, P2: 02, Lc: 02, 00, 00, Le: 02, 7f, ff, SW1: 90, SW2:
00
CLA: 80, INS: 13, P1: 01, P2: 02, Lc: 03, 00, 00, c0, Le: 40, 11, 22, 33, 44, 55
, 66, 77, 88, 20, 00, 11, 00, 00, 1d, 00, 90, 00, c2, 00, c1, 20, 00, 00, 00, 23
, 18, 00, c0, 00, bd, 01, 00, 20, 00, 00, 00, 3b, 18, 00, bd, 00, 0c, 00, 00, 80
, 00, 00, 00, 00, 1b, 00, 0a, a0, 00, 00, 00, 62, 03, 01, 0c, 01, 01, 2c, 00, SW
1: 90, SW2: 00
CLA: 80, INS: 15, P1: 01, P2: 02, Lc: 02, 00, 00, Le: 02, 00, 02, SW1: 90, SW2:
00
```

For `baload_bastore` POC, the region marked with colors illustrates READ_MEM APDU request, associated response and status bytes of APDU processing. The response data contains the result of reading card's memory through arbitrary object instance provided as an input to `baload` instruction.

By default, all APDU requests implemented by our code make use of the following APDU class value:

```
    private final static byte SEApplet_CLA       = (byte)0x80;
```

Additionally, each exploit applet instance has AID value of `0xa0:0x0:0x0:0x0:0x62:0x3:0x1:0xc:0x1`.

Table below provides more details with respect to APDU commands implemented by each POC. The table (along source codes) should be referenced for better understanding of Proof of Concept codes' outputs.

POC	INS	TYPE	DESCRIPTION
baload_bastore	0x10	PING	Check if applet code was successfully installed and is running *REQ APDU:* 00: `?? (unused)` 01: `?? (unused)` *RESP APDU:* 00: `0x12` 01: `0x34`
	0x11	STATUS	Check the status of the vulnerability (Issue 1) *REQ APDU:* 00: `?? (unused)` 01: `?? (unused)` *RESP APDU:* 00-3f: bytes of data read from a Cast object instance mimicing an array of bytes
	0x12	SETUP	Setup arbitrary memory reading condition by the means of a table of ints, Issue 1 is used to read card memory in a search for a header corresponding to a given table of ints, its header gets modified to indicate the length of 0x7fff *REQ APDU:* 00: `?? (unused)` 01: `?? (unused)` *RESP APDU:* 00-01: the result length of table of ints
	0x13	READ_MEM	Read memory through a table of ints *REQ APDU:* 00-01: offset to start reading data 02: length of data to read *RESP APDU:* 00-len: bytes of data read from a table of ints starting from given offset
	0x14	WRITE_MEM	Write memory through a table of ints *REQ APDU:* 00-01: offset to store val 02-05: val to store *RESP APDU:* 00-03: stored val
	0x15	CLEANUP	Cleanup exploit condition exploiting table of ints (store original table length to the header)

			REQ APDU: 00: `?? (unused)` 01: `?? (unused)` *RESP APDU:* 00-01: original length of table of ints
arraycopy	0x10	READ_MEM	Read memory with the use of a given array copy method *REQ APDU:* 00-01: offset to start reading data 02: length of data to read 03: type 00 - `arrayCopy` method 01 - `arrayCopyNonAtomic` method *RESP APDU:* 00-len: bytes of data copied from a Cast object instance mimicing an array of bytes
	0x11	WRITE_MEM	Write memory with the use of a given array copy method *REQ APDU:* 00-01: offset to start writing data 02: length of data to write 03: type 00 - `arrayCopy` method 01 - `arrayCopyNonAtomic` method 04-len: data bytes to write *RESP APDU:* 00-len: bytes of data copied to a Cast object instance mimicing an array of bytes
staticfield_ref	0x10	GET_STATIC	Read memory through a custom offset of an internal static field reference (provided at build time) *REQ APDU:* 00: `?? (unused)` *RESP APDU:* 00-01: the value read from a static field image
referencelocation	0x10	GETFIELD_A	Read memory through a custom token value of a `getfield_a` instruction, the target token value for all GETFIELD / PUTFIELD requests is provided at build time, it is chosen in such a way, so that it reflects header field corresponding to the length of a given array of ints *REQ APDU:* 00: `?? (unused)` *RESP APDU:* 00-01: the value read (current length of array of ints)
	0x11	PUTFIELD_A	Write memory through a custom token value of a `putfield_a` instruction, as the target reference value is checked, valid reference

			needs to be provided, in our case, making use of *this* is sufficient (its pointer value representation is sufficient to increase the length of a target array) *REQ APDU:* 00: `?? (unused)` *RESP APDU:* 00-01: the length of array of ints (the value of a field written by a `putfield_a` instruction)
	0x12	GETFIELD_B	Read memory through a custom token value of a `getfield_a` instruction (provided at build time) *REQ APDU:* 00: `?? (unused)` *RESP APDU:* 00-01: the value read
	0x13	PUTFIELD_B	Write memory through a custom token value of a `putfield_a` instruction, the target token value is chosen in such a way, so that header field corresponding to the length of a given array of ints gets overwritten by it *REQ APDU:* 00: `?? (unused)` *RESP APDU:* 00-01: the length of array of ints (the value of a field written by a `putfield_a` instruction)
	0x14	GETFIELD_S	Read memory through a custom token value of a `getfield_a` instruction (provided at build time) *REQ APDU:* 00: `?? (unused)` *RESP APDU:* 00-01: the value read
	0x15	PUTFIELD_S	Write memory through a custom token value of a `putfield_a` instruction, the target token value is chosen in such a way, so that header field corresponding to the length of a given array of ints gets overwritten by it[4] *REQ APDU:* 00: `?? (unused)` *RESP APDU:* 00-01: the length of array of ints (the value of a field written by a `putfield_a` instruction)
swap_x	0x10	TRIGGER_SWAPX	Trigger the invocation of a malformed

[4] the value to write needs to be a valid reference in order to avoid an exception, *this* pointer is used in our case as it sufficiently illustrates the flaw and still allows to increase the target array length (observed reference value for this was `0xc0` > original array length).

			swap_x instruction *REQ APDU:* 00: ?? (unused) *RESP APDU:* 00-01: 0x1234 value, but it is never received (JCRE crash is signaled with IP value set to 0x33445566)
nativemethod	0x10	NREAD_SHORT	Invoke native readShort method of NativeMethods class *REQ APDU:* 00-03: addr to read data from 04-05: off to read data from *RESP APDU:* 00-01: value returned by readShort method invoked for specific arguments
	0x11	NWRITE_SHORT	Invoke native writeShort method of NativeMethods class *REQ APDU:* 00-03: addr to write data to 04-05: off to write data to 06-07: val to write *RESP APDU:* 00-01: value stored to designated address - the result returned by readShort method invoked same arguments as write operation

During build, test.scr script is merged with generated applet installation scripts (install1.scr and install2.scr corresponding to package and applet install). The output of this process is stored as scripts\test.scr file. This is the file used as input to ApduTool.

REFERENCES

[1] JAVA CARD TECHNOLOGY
https://www.oracle.com/technetwork/java/embedded/javacard/overview/index.html

[2] JAVA CARD CLASSIC PLATFORM SPECIFICATION 3.0.5
https://www.oracle.com/technetwork/java/embedded/javacard/downloads/index.html

About Security Explorations

Security Explorations (http://www.security-explorations.com) is a security company from Poland, providing various services in the area of security and vulnerability research. The company came to life as a result of a true passion of its founder for breaking security of things and analyzing software for security defects. Adam Gowdiak is the company's founder and its CEO. Adam is an experienced Java Virtual Machine hacker, with over 100 security issues uncovered in the Java technology over the recent years. He is also the Argus Hacking Contest co-winner and the man who has put Microsoft Windows to its knees (the original discoverer of MS03-026 / MS Blaster worm bug). He was also the first expert to present a successful and widespread attack against mobile Java platform in 2004.

