Security Vulnerability Notice

SE-2019-01-ORACLE-3

[Security vulnerabilities in Java Card, Issues 26-32]

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 additional 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	
26	origin	`__checkMethod` code
	cause	insufficient checks for targets of code execution transfer instructions
	impact	execution of unverified bytecodes
	status	verified
27	origin	`_getLocalReference` code
	cause	no checks for local variable index
	impact	compromise of memory safety / arbitrary read access of card memory
	status	verified
28	origin	`_setLocalReference` code
	cause	no checks for local variable index
	impact	compromise of memory safety / arbitrary write access to card memory
	status	verified
29	origin	`_getLocalShort` code
	cause	no checks for local variable index
	impact	compromise of memory safety / arbitrary read access of card memory
	status	verified
30	origin	`_setLocalShort` code
	cause	no checks for local variable index
	impact	compromise of memory safety / arbitrary write access to card memory
	status	verified
31	origin	`_getLocalInt` code
	cause	no checks for local variable index
	impact	compromise of memory safety / arbitrary read access of card memory
	status	verified
32	origin	`_setLocalInt` code
	cause	no checks for local variable index
	impact	compromise of memory safety / arbitrary write access to card memory
	status	verified

Issues 26-32 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].

Issue 26 is due to an insufficient checking of methods' bytecodes by the CAP installer inside `__checkMethod` subroutine. Bytecode verification is conducted by it in a linear fashion rather than by following the real control flow. During this process, targets of all code execution transfer instructions[1] are expected to be within given method's range (between method start and end location). No check for these targets is however conducted with respect to bytecode granularity (different instruction lengths). As a result, it is possible to transfer execution into the middle of a bytecode instruction and execute unverified bytecode sequence embedded by its operand.

In our Proof of Concept code, we rely on specially crafted sequences of `iipush` bytecode instructions in order to achieve a given sequence of unverified code. Each `iipush` opcode

[1] such as conditional and unconditional jumps, subroutine jumps and exception handlers.

can be used to embed 1 or 2 bytecode instructions followed by a jump to the next `iipush` in the chain. This is illustrated on Fig. 1.

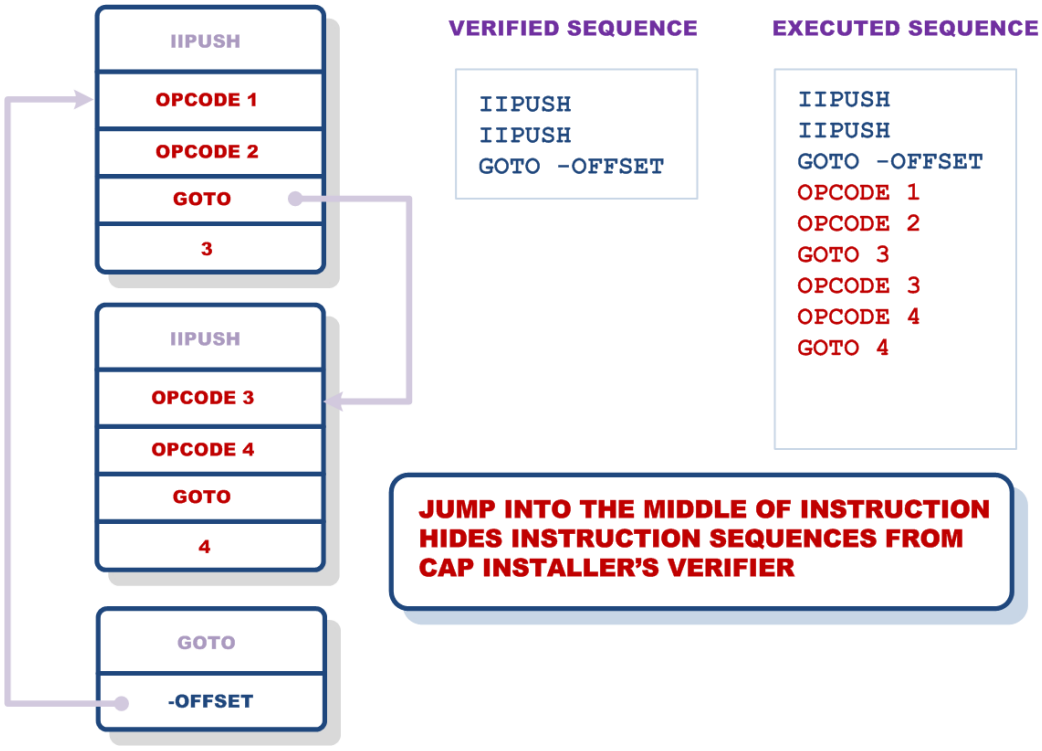

Fig. 1 Illustration of Issue 26 (a sequence of unverified instructions).

Issue 26 is not alone sufficient to compromise memory safety of a target Java Card VM. This can be however accomplished by combining it with one of the Issues 26-32.

Issues 26-32 are caused by no security checks conducted at runtime with respect to bytecode instructions conducting local variables' access (*sload*, *sstore*, *aload*, *astore*, etc.). There are several groups of these instructions implementing various local variable access (read or write and access short, reference or integer). These groups rely on a different vulnerable subroutine for given access implementation as indicated in a table below.

VULNERABLE SUBROUTINE	INSTRUCTION GROUP
_getLocalReference	getfield_a_this, getfield_b_this, getfield_s_this, getfield_i_this, putfield_a_this, putfield_b_this, putfield_s_this, putfield_i_this, **aload**, aload_0, aload_1, aload_2, aload_3, ret
_setLocalReference	**astore**, astore_0, astore_1, astore_2, astore_3
_getLocalShort	**sinc**, **sinc_w**, **sload**, sload_0, sload_1, sload_2, sload_3
_setLocalShort	**sinc**, **sinc_w**, **sstore**, sstore_0, sstore_1, sstore_2, sstore_3
_getLocalInt	**iinc**, **iinc_w**, **iload**, iload_0, iload_1, iload_2, iload_3
_setLocalInt	**iinc**, **iinc_w**, **istore**, istore_0, istore_1, istore_2, istore_3

Some of these instructions (indicated in red) can be encoded with arbitrary variable index pointing beyond the allowed stack location of a currently executing method. Upon proper arrangement of a stack layout and target local variable index, the content of saved methods' frames can be accessed (Fig. 2).

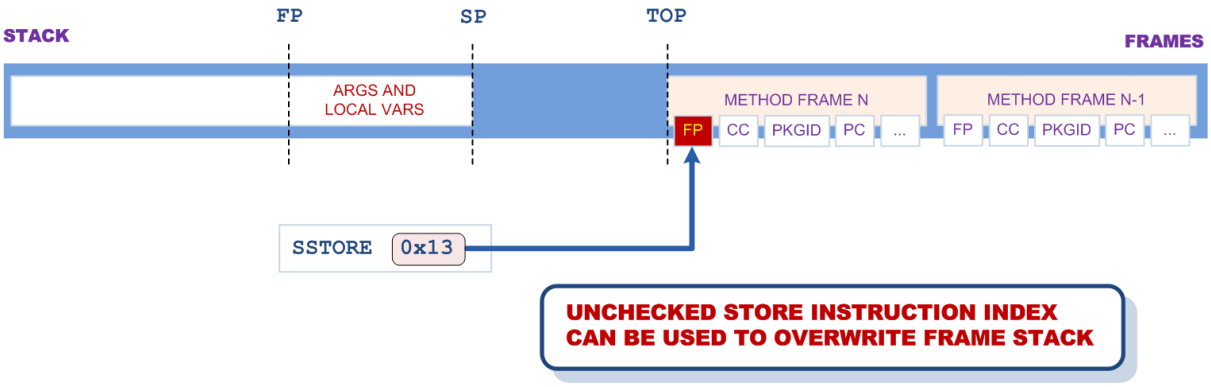

Fig. 2 Frame stack overwrite with the use of an unbounded local variable access instruction.

As a result, the frame pointer value (FP) denoting base stack location for methods' arguments and local variables can be changed to point to arbitrary memory address. Such a changed FP value can be further used to read card memory from within the method higher in a call stack (the one to which the return is made and which restores the overwritten FP value). This memory reading can be achieved by the means of bytecode instructions accessing local variables as they all rely on FP pointer.

The exploitation process implemented by our Proof of Concept code proceeds as following:

- `read_stack_frame_s` or `read_stack_frame_a` method is invoked recursively *N* number of times in order to decrease the gap between FP and Top pointer values, FP indicates local variables location and it is increased along every method call, Top pointer denotes the area where method frames are saved and its value decreases for every method call,
- method at call depth *N* exploits Issue 28 or Issue 30 for the change of a saved FP pointer value (FP value used by method at call depth *N-1)*, this change is implemented by the means of a store instruction[2] to variable location beyond current method's stack frame, Issue 26 is exploited in order to hide the target store instruction from the CAP file verifier,
- method at call depth *N-1* restores the value of a changed (denoting a user provided memory address) FP pointer, a local variable access results in a reading of a card memory through FP,
- method at call depth *N* (a dedicated call to static `store_val` method) stores read memory value into a static variable. It is not possible to simply return or store this result to any instance field at call depth *N-1* due to invalid FP pointer value, such an

[2] `astore` or `sstore` in our case.

operation can be conducted only by the method with valid SP and FP values[3] (enforced at a higher call level),
- method at call depth *N-2* cleans up the invalid FP pointer value (restore of the legitimate saved FP value), which preserves the code from a crash.

Table below provides more details with respect to APDU commands implemented by our Proof of Concept code illustrating the reported issues.

POC	INS	TYPE	DESCRIPTION
localvars	0x10	READ_MEM	Read card memory by the means of an overwritten FP pointer *REQ APDU:* 00-01: offset to start reading memory from 02: length of data to read 03: local variable access to exploit 00: `sload` / `sstore` bytecodes 01: `aload` / `astore` bytecodes *RESP APDU:* 00-len: bytes of data read from card memory starting at offset

Additionally, the Gen tool described in our initial report takes 2 arguments that correspond to the following:

- *arg0* - fixed value 6 (generation of a POC illustrating described issues),
- *arg1* - the local variable index to use for `sload` / `aload` instructions. It can be used to verify Issues 27 or 29 (by default the POC accesses card memory with the use of local variable access at index 0)

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

[3] bytecode instructions that trigger stack pop operation verify that SP and FP values are valid. Pop operation occurs for all `return`, `putfield` and `putstatic` instructions (their arguments are popped off the stack).

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.