Java Card Virtual Machine Compromising from a Bytecode Verified Applet

Julien Lancia[1] and Guillaume Bouffard[2]

[1] THALES Communications and Security S.A.S
Parc technologique du canal, Campus 2 – Bat.A
3 avenue de l'Europe, 31400 Toulouse, France
julien.lancia@thalesgroup.com
[2] Agence Nationale de la Sécurité des Systèmes d'Informations (ANSSI),
51, boulevard de La Tour-Maubourg, 75700 Paris 07 SP, France.
guillaume.bouffard@ssi.gouv.fr

Abstract. The Byte Code Verifier (BCV) is one of the most important security element in the Java Card environment. Indeed, embedded applets must be verified prior installation to prevent ill-formed applet loading. In this article, we disclose a flaw in the Oracle BCV which affects the applet linking process and can be exploited on real world Java Card smartcards. We describe our exploitation of this flaw on a Java Card implementation that enables injecting and executing arbitrary native malicious code in the communication buffer from a verified applet. This native execution allows snapshotting the smart card memory with OS rights.

Keywords: Java Card, Software Attack, BCV vulnerabilities

1 Introduction

Developing smart card applications is a long and complex process. Despite existing standardization efforts, *e.g.*, concerning power supply, input and output signals, smart card development used to rely on proprietary Application Programming Interfaces (APIs) provided by each manufacturer. The main drawback of this development approach is that the code of the application can only be executed on a specific platform, thus lowering interoperability.

To improve the interoperability and the security of embedded softwares, the Java Card technology was designed in 1997 to allow Java-based applications for securely running on smart cards and similar footprint devices. Due to the resources constraints of this device, only a subset of the Java technology was retained in the Java Card technology. The trade-offs made on the Java architecture to permit embedding the Java Card Virtual Machine (JCVM) on low resource devices concern both functional and security aspects.

1.1 The Java Card Security Model

In the Java realm, some aspects of the software security relie on the Byte Code Verifier (BCV). The BCV guarantees type correctness of the code, which in turn guarantees the Java properties regarding memory access. For example, it is impossible in Java to perform arithmetic operations on references. Thus, it must be proved that the two elements on top of the stack are bytes, shorts or integers before performing any arithmetic operation. Because Java Card does not support dynamic class loading, byte code verification is performed at loading time, *i.e.* before installing the Converted APplet (CAP) file onto the card. Moreover, most of Java Card platforms do not embed an on-card BCV as it is expensive in terms of memory consumption. Thus, bytecode verification is performed off-card, either directly by the card issuer if he controls the loading chain, or by a trusted third party that signs the application as a verification proof.

In addition to static off-card verification enforced by the BCV, the Java Card Firewall performs runtime checks to guarantee applets isolation. The Firewall partitions Java Card's platform into separated protected object spaces called contexts. Each package is associated to a context, thus preventing instances of a package from accessing (reading or writing) data of other packages, unless it explicitly exposes functionality through a Shareable Interface Object.

Despite all the security features enforced by the Java Card environment, several attack paths [1,2,4,5,6,10,11,13,14,15,19,22] have been found exploitable by the Java Card security community.

1.2 State-of-the-art on Java Card Byte Code Verifier flaws

The BCV is a key component of the Java Card platform's security. A single unchecked element in the CAP file, while apparently insignificant, can introduce critical security flaws in smart cards as shown in [11].

Although exhaustively testing a piece of software is a complex problem, several attempts have been made to characterize the BCV of the Java Standard Edition from a functional and security point of view. In [24], the authors rely on automatic test cases generation through code mutation and use a reference Virtual Machine (VM) implementation including a BCV as oracle. In [8], a formal model of the VM including the BCV is designed, then model-based testing is used to generate test cases and to assess their conformance to the model.

In the Java Card community, several works aim at providing a reference implementation of an off-card [16] or an on-card [3, 9] Java Card BCV. These implementations are mainly designed from a formal model and can be used to test the BCV implementation provided by Oracle. As for the VM, model-based testing approaches [7, 23] were used to assess on Java Card BCV implementations. As of today, no full reference implementation or model of the Java Card BCV has been proposed.

The Oracle's BCV implementation in version 2.2.2 was analyzed by Faugeron et al. [11]. In this implementation, the authors identified an issue in the branching instructions interpretation during the type-level abstract interpretation performed by the BCV. The authors exploited this issue to perform a type confusion in a local variable, undetected by Oracle's BCV. This issue in the BCV was patched by Oracle from version 3.0.3.

Since the version 3.0.3, no security flaw identification or exploitation in the Java Card BCV has been publicly signaled. In this paper, we present a new flaw discovered in the Java Card BCV from version 2.2.2 to 3.0.5 and we describe an exploitation of this flaw.

Section 2 introduces how a missing check in the Oracle's BCV implementation may allow an adversary to control a method offset and thus to trigger unverified bytecode execution. Section 3 shows how to success in exploiting this mechanism on a real Java Card product to trigger the execution of native code injected in a communication buffer. Finally, we evaluate our results on other Java Card products and propose a countermeasure to prevent the attack.

2 A Flaw in the BCV

2.1 The BCV Duty

The BCV enforces various security and consistency checks that guarantee each embedded application remains confined in its own sandbox. These verifications are performed on the CAP file, which is the binary representation of the classes that are loaded on the card. The BCV enforces two main kinds of checks: **type correctness** on the code and **structure verification** on the CAP file. The first one aims at performing an abstract interpretation of each method code in order to identify forbidden type conversion. The last one is an analysis of the CAP file structure to validate its consistency with the Java Card specification [20], and is detailed in the next section.

2.2 Verification of the CAP File Structure

The CAP file is composed of twelve different components, with internal and external dependencies, that are checked during the CAP file verification. Internal dependencies verification aims at validating the component properties as defined by the Java Card specification. External dependencies checks validate that redundant information specified in different components are compliant with each other. For example, each component has a `size` field that must be compliant with the `component-sizes` array contained in the Directory component where the sizes of every components are specified. An overview of all external dependencies between components in a CAP file are summarized in Figure 1 borrowed from [12].

Among the twelve components stored in the CAP file, we will focus on the following components:

- the Method component stores the code of all methods in the package, concatenated as a set of bytes;

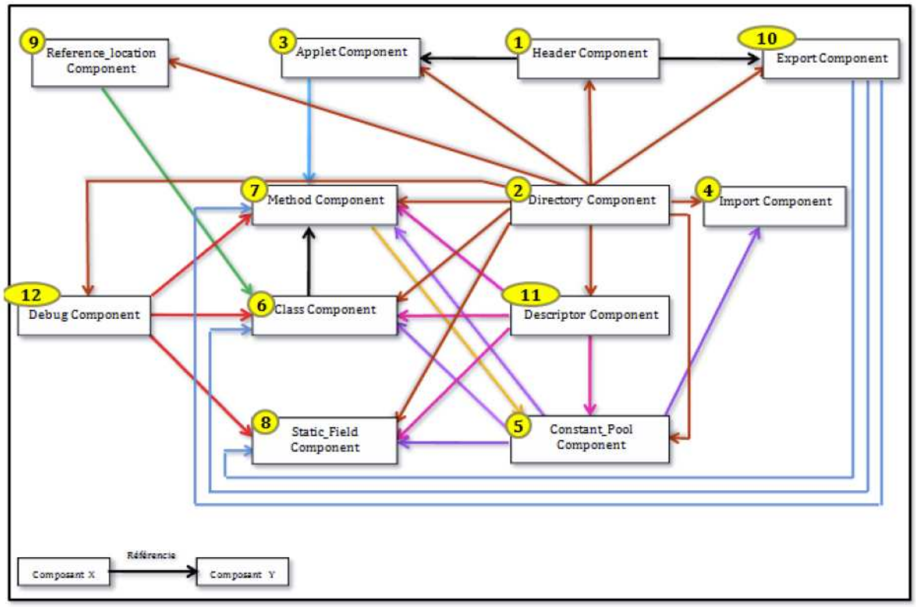


Fig. 1. External dependencies between components in a CAP file [12].

- the Constant Pool component contains an entry for each of classes, methods and fields referenced in the Method component;
- the Class component describes each classes and interfaces defined in the package, in a way that allows executing operations on that class or interface;
- the Descriptor component provides sufficient information to parse and verify all elements of the CAP file. This component is the main entry point for a byte code verification.

The Descriptor component is keystone of the BCV operations, but it has little or no importance for the card's processing and is therefore optionally provided during the loading of the applet.

Because of its purpose, the Descriptor component references several elements in the other components, and even provides redundant information with regards to these components. On the opposite, no component references the Descriptor component.

An analysis of the behavior of the BCV regarding the external dependency checks, and particularly the redundant information between components, brought us to identify a missing external dependency check between the Class component and the Descriptor component. We present the details of this BCV flaw and the resulting exploitation in the next sections.

2.3 Missing Check in the BCV

The missing check we have identified in the BCV involves the token-based linking scheme. This scheme allows downloaded software to be linked with API already embedded on the card. Accordingly, each externally visible item in a package is assigned a public token that can be referenced from another package. There are three kinds of items that can be assigned public tokens: classes, fields and methods. The bytecodes in the Method component refer to the items in the Constant Pool component, where the tokens required to perform the bytecode operation (*e.g.* class and method token for a method invoke) are specified.

When the CAP file is loaded on the card, the tokens are linked with the API and resolved from token form into the internal representation used by the VM. The linking process operates on the bytecode and is performed in several steps:

1. each token is an index in the Constant Pool component. The item stored at the provided index specifies the public tokens of the required items (*e.g.*, class and method token for a method invoke);
2. the tokens are resolved into the JCVM internal representation. For a method invoke, the class token identifies a `class_info` element in the Class component;
3. in the `class_info` element, the `public_virtual_method_table` array stores the methods internal representation. The method token is an index into the `public_virtual_method_table` array;
4. the element in the `public_virtual_method_table` at the method token index is an absolute offset in the Method component to the header and the bytecode of the method to execute.

The Figure 2 summarizes the linking process for a method call.

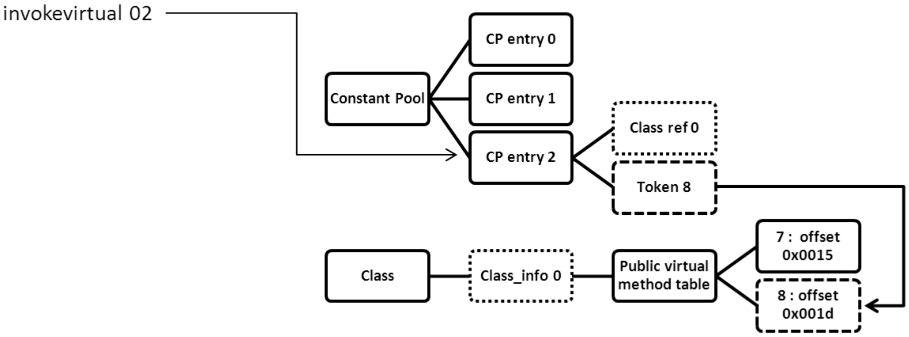


Fig. 2. Overview of the linking process for a method call.

The absolute offset in the Method component to the header and the bytecode of the method to execute is a redundant information in the CAP file as it is stored both in the `public_virtual_method_table` elements in the Class component and in the `method_descriptor_info` elements in the Descriptor component. The offset information in the Descriptor component is used exclusively by the BCV before loading, while the offset information in the Class component is used exclusively by the JCVM linker on card. Thus, any ill-formed offset information in the Class component remains undetected by the BCV checks, but is still used by the JCVM linker on card.

2.4 Exploiting the BCV flaw

As presented so far, the BCV flaw we expose allows manipulating the method offset information in the Class component while remaining consistent with the BCV checks. The exploitation of this flaw consists in deleting an entry in the `public_virtual_method_table` of a `class_info` element in the CAP file. The resolution of the corresponding method offset during the JCVM linking leads to an overflow in the Class component, as presented in Figure 3. This overflow brings the JCVM to interpret the content of the memory area following the Class component on card as a method index.

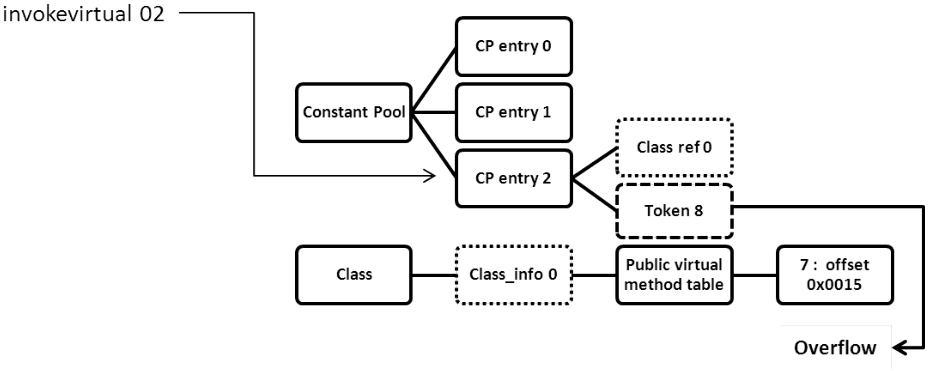


Fig. 3. Overflow in the linking process with for a method call.

The loading order of the CAP components is defined by the Java Card specification. This order specifies that the Method component is loaded right after the Class component. It is thus very likely that the Method component is stored next to the Class component in the card's memory. As a result, the Class component overflow is likely to fall into the Method component. In this eventuality, the offset of the method resolved in overflow is the numerical value of a bytecode in the Method component, that can be controlled by the applet developer. The Figure 4 presents an exploitation of the Class component overflow through the Method component.

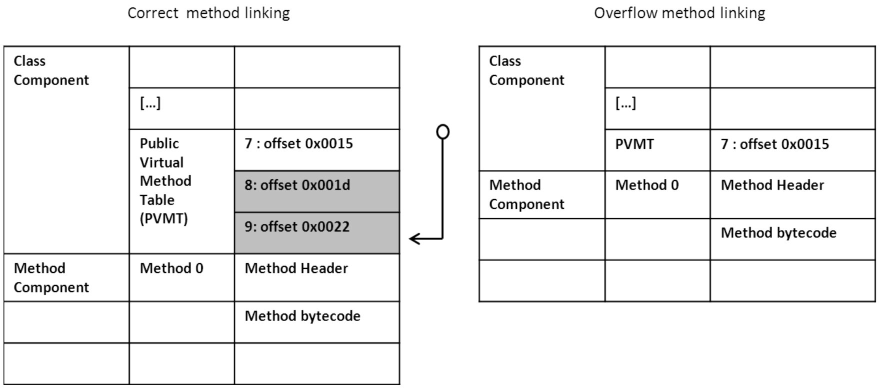


Fig. 4. Figure on left shows a successful linking in the Class component. Figure on the right shows the Class component overflow during linking when grayed out elements are deleted. Class component overflow falls into the Method component.

3 Code Injection from a Bytecode Verified Applet

In the previous section, we have presented, in the eventuality of a favorable memory mapping, how an attacker can exploit a BCV flaw to specify an arbitrary method offset in a BCV validated applet. In this section, we present the exploitation of this flaw on a real product that allows us to inject and execute native code in a communication buffer from a BCV validated applet.

The attack steps necessary to reach arbitrary native code on the Java platform are summed up hereafter and detailed in the next sections. First, we exploit the BCV flaw presented in Section 2 to forge an arbitrary method header in the Method component. This arbitrary method header is then used to abuse the native method execution mechanism of the platform and thus create a buffer overflow in the native method table. Finally, this buffer overflow allows dereferencing the communication buffer address as a native function. As a consequence, the data sent to our verified applet through the communication channel are executed as native code on the JCVM.

This full attack is a proof of concept to demonstrate that the flaw discovered in the Oracle BCV may jeopardize the security of Java Card smartcards.

3.1 Native Execution in the Virtual Machine

We validate the exploitation of the BCV flaw on an open Java Card platform embedded on an ARM micro-controller. This Java Card platform was provided in the context of a security expertise, thus both the code and the memory mapping of the VM were made available.

The runtime environment of this platform provides a mechanism that allows switching execution to native implementations of Java Card API methods for performance reasons. The implementation of this mechanism is similar to the Java Native Interface (JNI) mechanism provided in classical Java VMs [18].

In the JNI approach, the native methods are identified through a dedicated flag (`ACC_NATIVE`) in the method header. According to the JCVM specification, the native header flag is only valid for methods located in the card mask. Therefore a native method loaded in a CAP file is not compliant with the Java Card specification, and is thus rejected by the offcard verifier.

The native method resolution in JNI relies on *interface pointers*. An interface pointer is a pointer to a pointer. This pointer refers to an array of pointers, each one itself pointing on to an interface function. Every interface function is stored at a predefined offset inside the array. Figure 5 illustrates the organization of an interface pointer. The offset inside the array where the native function pointer is to be found is provided in the body of the native method.

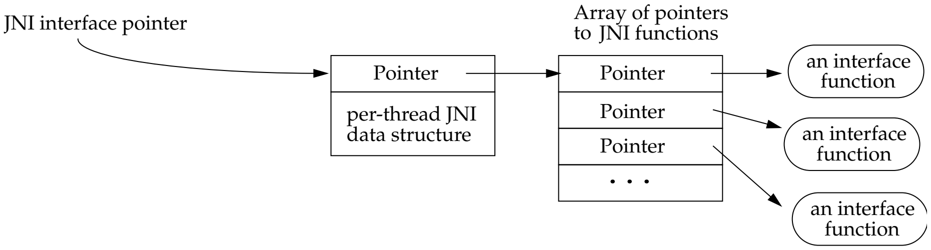


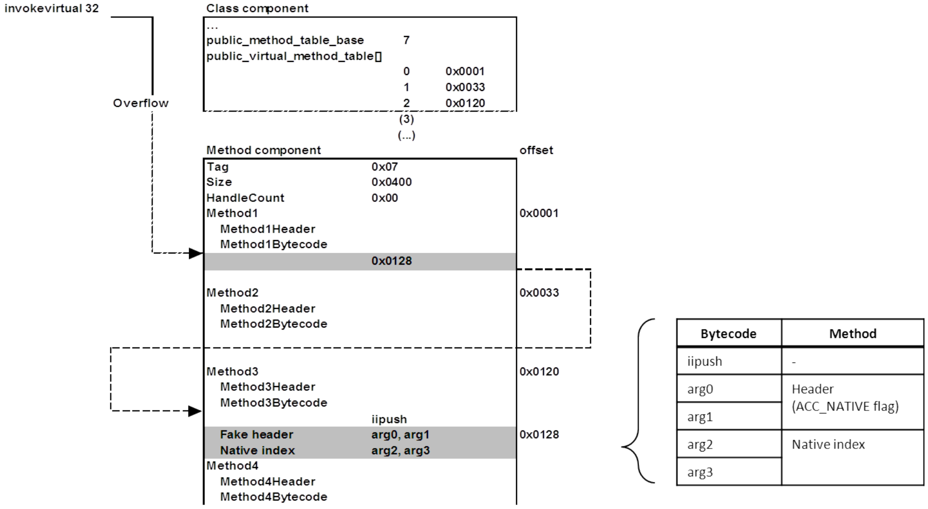
Fig. 5. JNI functions and pointers [17].

3.2 Native execution from a validated applet

As presented in Section 2 a missing check in the BCV can cause an overflow that brings the VM to resolve the method offsets outside the Class component. In the VM implementation we use to exploit our attack, the Class component overflow falls into the Method component so the value of the method offset can be specified as the numerical value of a bytecode in the Method component.

According to the JCVM specification [20], the offset of a method must point to a method header structure in the Method component, followed by the bytecode of the method. When exploiting the BCV flaw, the offset is controlled by the developer so it can point to any portion of the Method component. This can be used to make the method offset pointing on a portion of the bytecode that can be interpreted as a method header. The `iipush` bytecode can be used for this purpose, as its operand is a 4-bytes constant that is not interpreted by the BCV . This 4-bytes constant is thus used to code a method header containing the `ACC_NATIVE` flag and the native method index. This `iipush` bytecode is accepted by the BCV because it forms a valid bytecode sequence, but when the operand is interpreted as a native method header (through an overflow on the

Class component), the control flow switches to native execution. Figure 6 shows the attack path from the Class component overflow to the native execution of a JNI method.



Bytecode	Method
iipush	-
arg0	Header (ACC_NATIVE flag)
arg1	
arg2	Native index
arg3	

Fig. 6. Exploitation of the Class component overflow to execute a native method.

We were thus able, by specifying the adequate value for the method offset, to execute any of the native methods provided by the VM in the array of JNI native function pointers.

3.3 Abusing the Native Execution Mechanism for Code Injection

The attack so far allows calling JNI native methods provided by the platform, that are stored in an array of JNI native function pointers (or *native array*). When a native method call occurs, the switch from the Java runtime environment to the native execution environment requires an index in the native array to determine the native function pointer. Experimentation on the target JCVM allowed us to determine that an overflow on the native array can be achieved by specifying the relevant index in the native method body. Thus, any memory content stored next to the native array can be exploited as a native function pointer.

An analysis of the memory mapping of the product shows that a memory zone next to the native array contains a pointer to the communication buffer used for Host Controller Protocol (HCP) communications. The HCP protocol handles the transport layer of the Single Wire Protocol (SWP) protocol, involved in Near Field Communication (NFC) communications with smart cards. HCP messages encapsulates ISO7816 Application Protocol Data Unit (APDU) that are conveyed to the smartcard over SWP.

Using the overflow on the native array, we are able to use the HCP communication buffer pointer as native function pointer. The execution of this native function pointer leads to executing the content of the HCP communication buffer as a native assembly function.

The HCP protocol has several properties that limit the use of the HCP communication buffer as a native payload injection placeholder:

1. HCP packets are prefixed with a HCP message header and an HCP packet header. These headers are interpreted as native assembly opcodes.
2. HCP enforces fragmentation of messages, which limits packets size to 27 bytes. The entire native payload must thus be contained in 27 bytes.

In order to gain more space to inject our attack payload, we inject a minimal payload in the HCP communication buffer whose only purpose is to redirect the execution flow to the ISO7816 APDU buffer. This minimal redirection payload is presented in Table 1. Because the HCP communication buffer pointer is used as a function pointer, all the HCP buffer is interpreted as native code, including packet header, message header and encapsulated APDU header. These header bytes produce no side effect as shown in Table 1, which lets the redirection payload execute properly.

HCP message	Interpretation	Native code	Comment
`82 50`	Packet header Message header	`STR  r2, [r0, r2]`	No side effect
`00 10`	`CLA/INS`	`ASRS r0, r0, #0`	No side effect
`00 00`	`P1/P2`	`MOVS r0, r0`	No side effect
`14 00`	`Lc/padding`	`MOVS r4, r2`	No side effect
`E9 2D 5F FC`	`Data`	`PUSH {r2-r12, lr}`	
`F6 4A 54 D0`		`MOVW r4, #0xADD0`	
`F6 CA 54 D1`		`MOVT r4, #0xADD1`	`r4 = &apduBuffer`
`47 A0`		`BLX  r4`	`branch to apduBuffer`
`E8 BD 9F FC`		`POP  {r2-r12, pc}`	

Table 1. Native payload in the HCP buffer that redirects the execution flow to the APDU buffer. Relevant payload data is grayed out.

The ISO7816 protocol has broaden fragmentation constraints, which offers sufficient space for a full native payload injection. We present in Table 2 a full payload injected in the APDU buffer that branches to a low level read/write OS function. Because the start address execution is chosen from the HCP message buffer payload, the header bytes are skipped and the native execution starts at the `push` instruction (Table 2, 3rd row).

The payload initializes the source parameter to the first 4 bytes of the payload (Table 2, 2nd row), such that the reading address can be selected directly in the

	APDU	Interpretation	Native code	Comment
1	00 12 00 00 31	APDU Header		CLA/INS/P1/P2/Lc
2	B1 FA 15 00	Data		source reading address
3	2D E9 FF 5F		PUSH {r0-r12, lr}	
4	F6 4A 56 D0		MOVW r6, #0xADD0	
5	F6 CA 56 D1		MOVT r6, #0xADD1	r6 = apduBuffer
6	35 68		LDR r5, [r6,#0x00]	r5 = *apduBuffer
7	28 46		MOV r0, r5	
8	00 F1 09 00		ADD r0, r0, #0x6A	*dest: apduBuffer + 0x6A
9	D5 F8 05 10		LDR r1, [r5, #0x08]	*src: *(apduBuffer + 8)
10	4F F0 40 02		MOV r2, #0x40	length: 0x40
11	F6 4A 54 D2		MOVW r4, #0xADD2	
12	F6 CA 54 D3		MOVT r4, #0xADD3	r4 = *read_function_ptr()
13	A0 47		BLX r4	call method
14	BD E8 FF 9F		POP {r0-r12, pc}	

Table 2. Native payload in the ISO7816 APDU buffer that calls an OS function to read an arbitrary memory zone and copies the result to the APDU buffer. Relevant payload data is grayed out.

APDU. Then, it initializes the destination address (where the read bytes are copied) to the address of the APDU buffer following the payload, such that the read bytes are immediately available for sending back through the APDU buffer. Finally, it branches to the low level OS function that performs the reading operation. As a result, any physical address of the card can be accessed through this native payload.

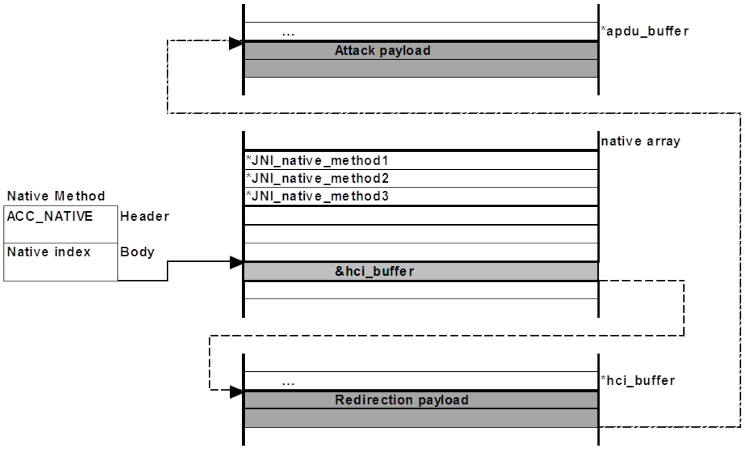


Fig. 7. Exploitation of the native array overflow to execute native code in the APDU buffer.

Figure 7 shows the execution flow from the native array overflow to the redirection payload in the HCP message buffer to the final attack payload in the APDU buffer. We were able to integrally dump the card memory and to reverse it using commercial reversing tools. The reversed code was identified as the code of the embedded JCVM.

4 Other Experimental Results

To evaluate the consequence of the BCV flaw on a broader range of virtual machine implementations, we tested, on different smart cards from different manufacturers, how much each of them supports the installation of an ill-formed applet. We evaluated seven cards from three distinct manufacturers (a, b and c). Each card name is associated with the manufacturer reference and its Java Card specification [20]. The list of evaluated Java Card smart cards is presented in Table 3.

Reference	Java Card Platform	GlobalPlatform Version	Details
`a-22a`	2.2.1	2.1.1	36 kB EEPROM, RSA
`a-22b`	2.2.2	2.1.2	80 kB EEPROM, RSA
`a-30c`	3.0.4	2.2.1	80 kB EEPROM, ePassport
`b-30a`	3.0.1	2.2.1	1 MB Flash memory, (U)SIM
`c-21a`	2.1.1	2.0.1	128 kB EEPROM, SIM
`c-21b`	2.1.1	2.0.1	64 kB EEPROM, RSA, AES
`c-22c`	2.2.2	2.2.2	256 kB Flash memory, (U)SIM

Table 3. Cards used during this evaluation.

None of the evaluated card implements an embedded BCV. On each card, an ill-formed applet is installed and, if the installation succeeds, the applet is executed. The ill-formed applet has a dereferenced method in the public virtual method table. Table 4 sums up the cards reactions.

Reference	Statut
`a-22a`	PCSC error: card mute. ✗
`a-22b`	PCSC error: card mute. ✗
`a-30c`	PCSC error: card mute. ✗
`b-30a`	No error: the card return the value `0x0701`. ✗
`c-21a`	Global platform error: error during the loading process (*applet rejected*). ✓
`c-21b`	Global platform error: error during the loading process (*applet rejected*). ✓
`c-22c`	Global platform error: error during the loading process (*applet rejected*). ✓

Table 4. Statuts of each evaluated cards.

As shown in Table 4, cards react differently to the ill-formed CAP file installation and execution. The cards with the symbol (✓) detect the ill-formed CAP file during the installation and reject it. On the other cards, marked with the symbol (✗), installation and execution succeed.

Successful executions cause either unexpected card response or card mute. Unexpected card response indicates that unexpected code execution occurred. Card mute may result from infinite loop or card's reaction to illegal code, which also indicates unexpected code execution.

These behaviors proof that the control flow of the JCVM is modified. We can thus conclude that the BCV flaw presented in this article can be exploited on a range of different Java Card smartcards.

The full attack path that results in arbitrary native code execution requires information about the memory mapping and the JCVM implementation that were not available for these tests. Therefore, we did not attempt to reproduce the full attack path.

5 Conclusion, Countermeasure and Future Works

We show in this article how a missing check in the Oracle's BCV implementation can be exploited on a Java Card. We demonstrated that this BCV issue has a critical impact on smart cards security through a proof of concept exploitation on a JCVM implementation. We have successfully managed to inject and execute native code in a communication buffer, and finally gain full read/write OS privileges on the whole card memory. Finally, we evaluated on a range of different cards from different manufacturers that most of the JCVM implementations do not protect themselves against the BCV issue exploitation.

Following our responsible disclosure of the BCV issue to Oracle, we were allowed to publish this article and a new version of the BCV was released[3]. This new BCV version detects the Class component inconsistency and thus mitigate our attack. A loading process including mandatory bytecode verification step with the latest Oracle's BCV provides a valid countermeasure against the attack presented in this paper.

With the identification of a new flaw in the Oracle's BCV implementation, one sees that the BCV must be entirely verified to lower the risks of new vulnerabilities disclosure. To reach this objective, an effort should be done to specify the security and functional requirements a BCV must comply with in order to protect JCVM implementations against software attacks.

References

1. Barbu, G., Duc, G., Hoogvorst, P.: Java card operand stack: Fault attacks, combined attacks and countermeasures. In: Prouff [21], pp. 297–313

[3] The BCV included in the Java Card SDK 3.0.5u1 prevents the introduced attack. This version was released on 19 August 2015.

2. Barbu, G., Thiebeauld, H., Guerin, V.: Attacks on java card 3.0 combining fault and logical attacks. In: Smart Card Research and Advanced Application, 9th IFIP WG 8.8/11.2 International Conference, CARDIS 2010, Passau, Germany, April 14-16, 2010. Proceedings. pp. 148–163 (2010)
3. Berlach, R., Lackner, M., Steger, C., Loinig, J., Haselsteiner, E.: Memory-efficient On-card Byte Code Verification for Java Cards. In: Proceedings of the First Workshop on Cryptography and Security in Computing Systems. pp. 37–40. CS2 '14, ACM, New York, NY, USA (2014)
4. Bouffard, G.: A Generic Approach for Protecting Java Card Smart Card Against Software Attacks. Ph.D. thesis, University of Limoges, Limoges, France (Oct 2014)
5. Bouffard, G., Iguchi-Cartigny, J., Lanet, J.: Combined software and hardware attacks on the java card control flow. In: Prouff [21], pp. 283–296
6. Bouffard, G., Lanet, J.: The ultimate control flow transfer in a java based smart card. Computers & Security 50, 33–46 (2015)
7. Calvagna, A., Fornaia, A., Tramontana, E.: Combinatorial interaction testing of a java card static verifier. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, Workshops Proceedings, March 31 - April 4, 2014, Cleveland, Ohio, USA. pp. 84–87. IEEE Computer Society (2014)
8. Calvagna, A., Tramontana, E.: Automated Conformance Testing of Java Virtual Machines. In: Barolli, L., Xhafa, F., Chen, H., Gómez-Skarmeta, A.F., Hussain, F. (eds.) Seventh International Conference on Complex, Intelligent, and Software Intensive Systems, CISIS 2013, Taichung, Taiwan, July 3-5, 2013. pp. 547–552. IEEE Computer Society (2013)
9. Casset, L.: Development of an Embedded Verifier for Java Card Byte Code Using Formal Methods. In: Eriksson, L., Lindsay, P.A. (eds.) FME 2002: Formal Methods - Getting IT Right, International Symposium of Formal Methods Europe, Copenhagen, Denmark, July 22-24, 2002, Proceedings. Lecture Notes in Computer Science, vol. 2391, pp. 290–309. Springer (2002)
10. Faugeron, E.: Manipulating the frame information with an underflow attack. In: Francillon, A., Rohatgi, P. (eds.) Smart Card Research and Advanced Applications - 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers. Lecture Notes in Computer Science, vol. 8419, pp. 140–151. Springer (2013)
11. Faugeron, E., Valette, S.: How to hoax an off-card verifier. e-smart (2010)
12. Hamadouche, S.: Étude de la sécurité d'un vérifieur de Byte Code et génération de tests de vulnérabilité. Master's thesis, University M'Hamed Bougara of Boumerdes, Faculty of Sciences, LIMOSE Laboratory, 5 Avenue de l'indépendance, 35000 Boumerdes, Algeria (2012)
13. Hamadouche, S., Bouffard, G., Lanet, J.L., Dorsemaine, B., Nouhant, B., Magloire, A., Reygnaud, A.: Subverting byte code linker service to characterize java card api. In: Seventh Conference on Network and Information Systems Security (SAR-SSI). pp. 75–81 (May 22rd to 25th 2012)
14. Hamadouche, S., Lanet, J.: Virus in a smart card: Myth or reality? Journal of Information Security and Applications 18(2-3), 130–137 (2013)
15. Lancia, J.: Java card combined attacks with localization-agnostic fault injection. In: Mangard, S. (ed.) Smart Card Research and Advanced Applications - 11th International Conference, CARDIS 2012, Graz, Austria, November 28-30, 2012, Revised Selected Papers. Lecture Notes in Computer Science, vol. 7771, pp. 31–45. Springer (2012)
16. Leroy, X.: Bytecode verification on Java smart cards. Softw., Pract. Exper. 32(4), 319–340 (2002)

17. Liang, S.: The Java Native Interface: Programmer's Guide and Specification. Addison-Wesley Professional, first edition edn. (Jun 1999)
18. Lindholm, T., Yellin, F., Bracha, G., Buckley, A.: The Java Virtual Machine Specification. Java Series, Addison-Wesley (May 2014)
19. Mostowski, W., Poll, E.: Malicious code on java card smartcards: Attacks and countermeasures. In: Grimaud, G., Standaert, F. (eds.) Smart Card Research and Advanced Applications, 8th IFIP WG 8.8/11.2 International Conference, CARDIS 2008, London, UK, September 8-11, 2008. Proceedings. Lecture Notes in Computer Science, vol. 5189, pp. 1–16. Springer (2008)
20. Oracle: Java Card 3 Platform, Virtual Machine Specification, Classic Edition. No. Version 3.0.5, Oracle, Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065 (2015)
21. Prouff, E. (ed.): Smart Card Research and Advanced Applications - 10th IFIP WG 8.8/11.2 International Conference, CARDIS 2011, Leuven, Belgium, September 14-16, 2011, Revised Selected Papers, Lecture Notes in Computer Science, vol. 7079. Springer (2011)
22. Razafindralambo, T., Bouffard, G., Lanet, J.: A friendly framework for hidding fault enabled virus for java based smartcard. In: Cuppens-Boulahia, N., Cuppens, F., García-Alfaro, J. (eds.) Data and Applications Security and Privacy XXVI - 26th Annual IFIP WG 11.3 Conference, DBSec 2012, Paris, France, July 11-13,2012. Proceedings. Lecture Notes in Computer Science, vol. 7371, pp. 122–128. Springer (2012)
23. Savary, A., Frappier, M., Lanet, J.: Detecting Vulnerabilities in Java-Card Bytecode Verifiers Using Model-Based Testing. In: Johnsen, E.B., Petre, L. (eds.) Integrated Formal Methods, 10th International Conference, IFM 2013, Turku, Finland, June 10-14, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7940, pp. 223–237. Springer (2013)
24. Sirer, E.G.: Testing Java Virtual Machines. In: International Conference on Software Testing And Review. San Jose, California (Nov 1999)