PhiAttack
Rewriting the Java Card Class Hierarchy

Jean Dubreuil[1] and Guillaume Bouffard[2,3](✉)

[1] Serma Safety & Security, Pessac, France
`j.dubreuil@serma.com`
[2] National Cybersecurity Agency of France (ANSSI), Paris, France
`guillaume.bouffard@ssi.gouv.fr`
[3] Information Security Group, DIENS École Normale Supérieure, CNRS, PSL University, Paris, France

Abstract. Compiling Java Card applets is based on the assumption that `export` files used to translate Java class item to Java Card CAP tokens are legitimate. Bouffard *et al.* [2] reversed the translation mechanism. Based on malicious Application Programming Interface (API) embedded in a target, they succeeded in making a man-in-the-middle attack where cryptographic keys can leak.

In this article, we disclose that, on a pool of legitimate `export` files, Java Card Virtual Machine (JCVM) implementations can be confused by a CAP file verified by the Java Card Bytecode Verifier (BCV). The disclosed vulnerability leads to Java Card class hierarchy rewriting. The introduced vulnerability is exploitable up to Java Card 3.0.5. Recently, Java Card 3.1.0 provides a new `export` file format which prevents this vulnerability.

Keywords: Java Card · BCV · Inheritance tree

1 Introduction

Java Card platform [14] is the most used technology embedded in secure components [13]. Java Card is a lightweight version of Java for resource-constrained devices as secure components. Therefore, such secure component embeds a virtual machine, which interprets application bytecodes already romized with the operating system or downloaded after issuance. Due to security reasons, the ability to download code into the card is controlled by a protocol defined by GlobalPlatform [7].

To build a Java Card application, an image of the targeted Java Card Virtual Machine (JCVM) implementation is required. This image gives information about the available Application Programming Interface (API) and the class hierarchy. In this article, we focus on how class inheritance is translated during the compilation process and loaded in a JCVM platform. We show this process can be corrupted to redefine the class-tree hierarchy which leads to execute malicious code.

1.1 Java Card Security Model

To install an applet on the Java Card platform, one must implement it in Java language and then build it within Java compiler (`javac`) to obtain Java `class` files. Those `class` files are not designed to be embedded in a resource-limited device. Indeed, the Java `class` files are executed as is by the Java Virtual Machine (JVM) where references are resolved by name; it is very costly in both execution time and memory space. The translated Java `class` files are named the CAP (for Converted APplet) file.

To run a Java applet on resource-constraint devices, the adopted solution is to translate reference name to token during a step made by the Java Card converter[1]. If the `class` file to convert implements features that can be used by other applications, a Java Card `export` file is also generated. The `export` file contains, for each Java reference name element, the associated token embedded on the device. Therefore, `export` files are also used by the bytecode converter during the translation process. After this translation, Java Card files are checked by the Bytecode Verifier (BCV) which statically verifies the compliance to the Java Card security rules. There is a unique CAP file by converted package, and it is signed to ensure its integrity and authenticity. On the device, the GlobalPlatform [7] layer verifies the CAP file signature. This part is described on the left part of Fig. 1.

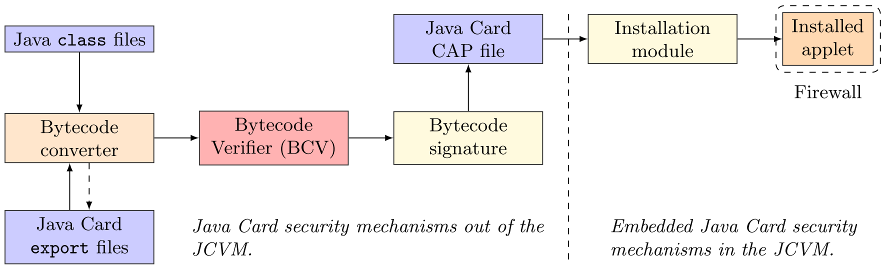


Fig. 1. Java Card security model.

After obtaining and signing the applet CAP file, the applet developer needs GlobalPlatform loading keys to load his applet or library. During the installation process, on the right part of Fig. 1, an embedded security module checks some security elements. The installed applet runs in its context segregated by the Java Card Firewall. It ensures that applet accesses only its data or specific shared features.

[1] The Java Card converter is included in the Java Card SDK available on the Oracle's website: https://www.oracle.com/fr/java/technologies/java-card-tech.html.

1.2 State-of-the-Art Java Card Platform Security

The Java Card platform implementation security has been thoroughly studied against software [1,3–5,8–10,12,17] attacks. Those attacks are implementation dependent and they are prevented by a BCV. The Java Card protection profile [15] requires the usage of a BCV to check the applet compliance from Java security rules.

How the BCV checks CAP files has been analysed in [6,11]. Lancia *et al.* [11] shows that the BCV does not verify the correctness of information stored twice in the CAP file. Based on this missing check, they succeed in breaking the JCVM sandbox by executing ill-formed bytecodes from BCV verified applet. This vulnerability was corrected in the BCV provided in the Java Card 3.0.5u3 toolchain.

To check the correctness of CAP files, the BCV analysis relies on `export` files used during the CAP file conversion. If an `export` file contains wrong information – information which does not correspond to the targeted JCVM – a vulnerability may occur. Disclosed by Mostowski *et al.* [12], using wrong `export` files, they succeed in making a type confusion upon a BCV-verified applet. Moreover, Bouffard *et al.* [2] succeed a Man-in-the-Middle attack based on malicious `export` files to extract cryptographic secrets. In their attack, they must install a backdoored API on a targeted JCVM and provide `export` files to link applet to this malicious API. Those `export` files replace the Java Card cryptographic API. On the targeted JCVM platform, the backdoored API makes interface with the legitimate Java Card cryptographic API and saves each key generated. However, this attack is interesting but hard to realize in practice: the attacker must force its victim to use corrupted `export` files whereas it is expected that any application developer use `export` files from Oracle's development kit.

1.3 Contribution

In this article, we generalise Bouffard *et al.*'s work [2] where we corrupt the Java Card class hierarchy. We succeed in confusing the CAP file import mechanism to force the targeted JCVM platform to use our Java class hierarchy instead of the legitimate one. As token resolution relies on runtime verification, our attack is not detected by a BCV. Therefore, we exploit the token resolution mechanism to execute malicious code on JCVM platform where each installed CAP file are checked by an up-to-date BCV.

Our contribution has been initially performed on Java Card specification 3.0.5 [14] as there is no publicly known product implementing a higher specification version. Therefore, in the paper, we use the BCV provided by the Java Card SDK 3.0.5.

We notice that the latest available Java Card SDK is the `3.1.0u5` version [16]. However, when writing this article, there is not product that implements this version.

This article is organized as follows: Sect. 2 describes the Java Card import mechanism in order to introduce the PhiAttack explained in Sect. 3. A discussion on how to counteract this attack is in Sect. 4. Section 5 concludes this article.

2 Java Card Import Mechanism

This section explains how imported packages are referenced in CAP and `export` files in order to introduce the exploited vulnerability.

When an application needs to call some methods from an external API, for instance the Java Card standard API, runtime must first import the package or the class containing this method. Importing classes and packages in Java Card is performed similarly to Java standard syntax as shown in Listing 1.1.

Listing 1.1. `SimpleImportExample` class description.

```
 1 package simple;
 2 
 3 import javacard.framework.*; // Importing the whole package
 4 import javacard.framework.JCSystem; // Importing only one class
 5 
 6 public class SimpleImportExample {
 7   public    byte publicField;
 8   private   byte privateField;
 9   protected byte protectedField;
10             byte packageField;
11 
12   public static short getVersionExample() {
13     // Use one of the imported features
14     return JCSystem.getVersion();
15 } }
```

As explained in Sect. 1.1, outputted by the Java Card toolchain, the CAP file contains application information to be executed as is by the JCVM. The `export` file has everything required to use public features provided to other applications. Therefore, the `export` file shares public application names and associated tokens.

2.1 Import Mechanism from the CAP File Point of View

The CAP file contains information to call the external methods. We now focus on `JCSystem.getVersion()` method (Listing 1.1, line 14) to understand CAP file import mechanism.

The `Import` and the `ConstantPool` components are used by the `Method` component when calling the `JCSystem.getVersion()` method as shown in Listing 1.2.

Listing 1.2. A `simple.cap` file partial view.

```
Import Component
  A0000000620001 // java/lang
  A0000000620101 // javacard/framework

ConstantPool Component
  // 0
  staticMethodRef 0.0.0()V; // java/lang/Object.<init>()V
  // 1
  staticMethodRef 1.8.9()S; // javacard/framework/JCSystem.getVersion()S

// ...

Method Component

  .method public static getVersionExample()S 1 {
```

```
16      .stack 1;
17      .locals 0;
18
19      L0: invokestatic 1 // javacard/framework/JCSystem.getVersion()S
20      sreturn;
21    }
```

In the `Import` component (Listing 1.2, lines 1 to 3), two packages are listed: `java.lang`, indexed at `0` and `javacard.framework`, indexed at `1`. Even if not explicitly imported in the Java source file, the `java.lang` package is automatically imported by the compiler.

All the imported packages are referenced by their corresponding Application Identifier (AID) value. In the `ConstantPool` component, the `JCSystem.getVersion()` method is referenced in the second entry, Listing 1.2, line 9. Value `1.8.9` is interpreted as followed:

– 1 represents the second imported package (there, `javacard.framework`),
– `8` represents the class token (`JCSystem`)
– and `9` the method token (`getVersion()`).

Class and method tokens are defined in the `export` file of `javacard.framework` package. Finally, the `invokestatic` bytecode references the second entry of the `ConstantPool` component, indicating to the JCVM where it can find the method to call.

2.2 Import Mechanism from the `export` File Point of View

Considering Listing 1.1, the obtained `export` file contains the declaration of:

– the `SimpleImportExample` class and reference to its super classes (in this case, only `Object` class),
– the `publicField` and `protectedField` fields. The `export` file contains: `public`, `protected`, `static` and `final` field declarations
– and the `getVersionExample()` method. As well as the fields, the `export` file contains `public`, `protected`, `static` and `final` method declarations.

In this example, we have seen that `export` file does not list the imported `javacard.framework` package. However, a package can publicly expose features that it had previously imported. This happens, for instance, when inheriting and in this case, the `export` file will trace the imported packages.

Listing 1.3. The `InheritingImportExample` class.

```
package inheriting;

import javacard.framework.ISOException;

public class InheritingImportExample extends ISOException {
  public InheritingImportExample(short reason) {
    super(reason);
} }
```

In Listing 1.3, a class inheriting from `ISOException` is defined at line 5. After converting this class, the CAP file will import the `javacard.framework` package as explained for Listing 1.2, based on the AID value. The `export` file will contain supplementary information because the `InheritingImportExample` class exposes all the public tokens from the `ISOException` class. Therefore, the following items are found in the `export` file:

– all the super classes of `InheritingImportExample`: in order, we have:

  1. `ISOException`,
  2. `CardRuntimeException`,
  3. `RuntimeException`,
  4. `Exception`,
  5. `Throwable`
  6. and `Object`.

– all the inherited public methods from these classes: `setReason()`, `getReason()` and `equals()`.

Unlike in the CAP file, imported tokens in `export` file are referenced using their fully qualified names. For instance, the `ISOException` class is defined by the `ConstantPool` entry shown in Listing 1.4.

Listing 1.4. Partial view of `inheriting` package `export` file.

```
tag    :  01 (cp_utf8_info)
length:  00 1f
utf8_bytes[]: javacard/framework/ISOException
```

In this Section we have seen how imported packages are referenced in CAP and `export` files. In some cases, the imported package is simultaneously defined in both files. However, an asymmetry exists as the CAP file references imported packages from their AID values while the `export` file references them using their fully qualified names.

The BCV may not be able to ensure that the AID used in the CAP file corresponds to the package name used in the `export` file and this may lead to inconsistencies as explained in the next Section.

3 PhiAttack

On Java Card platforms, every package is identified by a unique AID value. Actually, nothing prevents an application developer to create its own package with the same name as an already existing package, as long as the assigned AID value to this package is not already used by another one. At compilation and runtime, this is accepted: the BCV is able to identify and discriminate the two packages ensuring that the packages are properly used and the JCVM interprets bytecode from the content of CAP files that import packages with their AID.

3.1 Setting-up the Attack

Let's consider an application developer that creates two packages, both named `library` but each one has a different AID, as shown in Listing 1.5 and Listing 1.6. Each package contains a class, named `Phi`, with a method named `doSomething()`. However, this method signature is different from one package to another. The difference is highlighted in **red and underline**.

Listing 1.5. `library` package with `DEADBEEF01` AID.

```
package library;

public class Phi {
  public void doSomething() {
    // ...
} }
```

Listing 1.6. `library` package with `DEADBEEF02` AID.

```
package library;

public class Phi {
  public short doSomething() {
    // ...
} }
```

In the Java source code, one cannot import both versions of the `library` package at the same time. As each package has the same name, the compilation process cannot distinguish one from the other. However, this can be achieved by forging a CAP file that imports these packages, from their AIDs. Even if such a construction cannot be obtained in a common way, it will be, however, accepted by the BCV. In this case, the BCV properly handles the two packages and it is able to differentiate the two `Phi` classes. Such a construction is quite weird but is actually allowed.

A third package named `proxy` is described in Listing 1.7. It imports `library` package. At compilation time, only the `library` package defined in Listing 1.5 is given to the Java Card toolchain. Therefore, the CAP file of `proxy` package imports `library` with `DEADBEEF01` AID. The `PhiProxy` class only inherits from the `Phi` class. Therefore, the `export` file of `proxy` package references the `library.Phi` class and the `doSomething()` method with the correct signature (return type is `void`).

Listing 1.7. `proxy` package

```
package proxy;

import library.*;

public class PhiProxy extends Phi {}
```

A last package, named `exploit`, is created and described in Listing 1.8. This package imports two packages: `library` and `proxy`. At compilation time, the `library` package defined in Listing 1.6 is provided at the Java Card toolchain. Therefore, the CAP file of `exploit` imports `library` with `DEADBEEF02` AID. In `doExploit()` method, Listing 1.8 line 7, an instance of `PhiProxy` is created and its reference is saved in a variable of type `Phi`. Finally, the `doSomething()` is called.

Listing 1.8. `exploit` package

```
 1 package exploit;
 2 
 3 import library.*;
 4 import proxy.*;
 5 
 6 public class Exploit {
 7   public void doExploit() {
 8     PhiProxy proxyInstance = new PhiProxy();
 9     Phi phiInstance = proxyInstance;
10     short result = phiInstance.doSomething();
11 } }
```

Figure 2 shows the UML diagram of these packages in order to synthesise a global view of the dependencies between them.

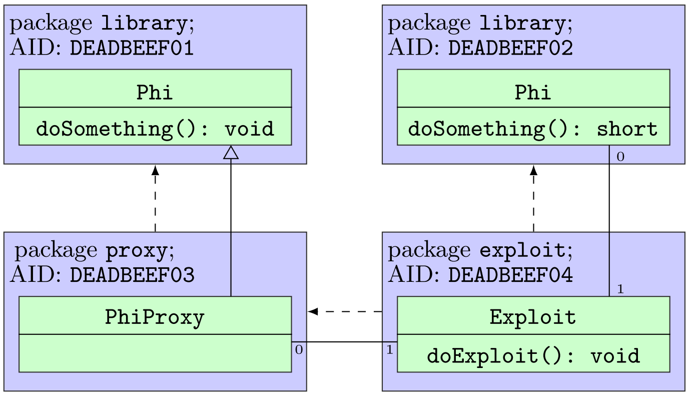


Fig. 2. UML diagram of PhiAttack.

The four packages described in this section are checked by the BCV. The obtained result is: `0 error and 0 warning`.

3.2 Understanding PhiAttack

Two processes must be studied here, 1) the analysis performed by the BCV on CAP and `export` files of `exploit` package and 2) the execution flow of `doExploit()` method at runtime.

On the one hand, to verify the `exploit.cap` file, BCV makes checks as introduced in Fig. 3. In the `doExploit` method in Listing 1.8, three parts are critical:

1. At line 8, an instance of `PhiProxy` is created. The obtained reference is stored in a variable of the same type. On Fig. 3, the BCV checks this instruction `new 0` by resolving token `0` (①) and reads the `ConstantPool` entry `0` to obtain `proxy.PhiProxy` type (②) in `proxy` `export` file.
2. At line 9 the previously stored reference is copied in a variable of type `Phi`. The compiler translates this operation by `aload` and `astore` instructions and

it does not insert `checkcast` instruction as `PhiProxy` type is a sub-class of `Phi`. From the BCV, type is ensured; `aload` instruction pushes a `PhiProxy` instance on operand stack and `astore` instruction pops a `Phi` instance from operand stack. There, the BCV validates the operation because it finds the mother class `Phi` from `library` package with `DEADBEEF02` AID. At this state of the verification, the BCV cannot know that the actual mother class is in package with `DEADBEEF01` AID. At runtime, as there is no `checkcast` instruction, no cast verification is performed.

3. At line 10, the `doSomething()` method is called on an instance of type `Phi`. On Fig. 3, to call this method, the `invokevirtual 1` instruction is checked by the BCV. To verify this method call, the BCV resolves token `1` (③). to obtain `library.Phi.doSomething()` method signature (④) in `library` `export` file.

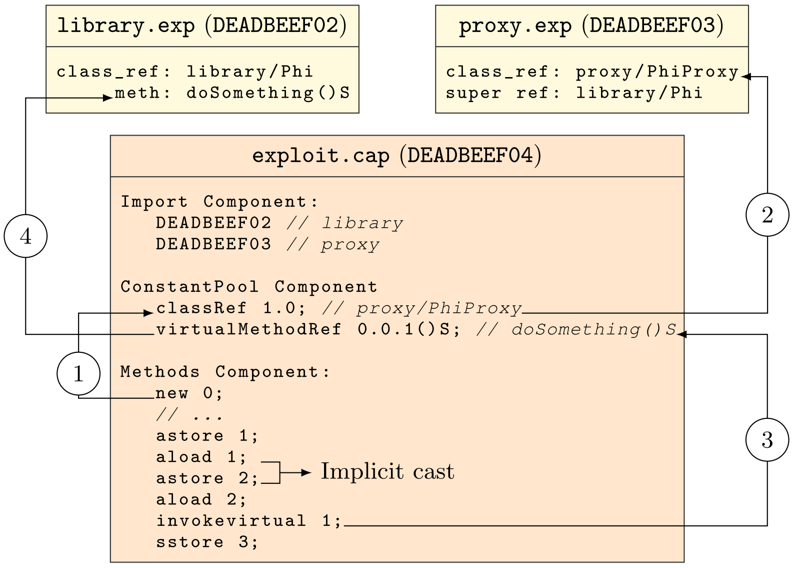
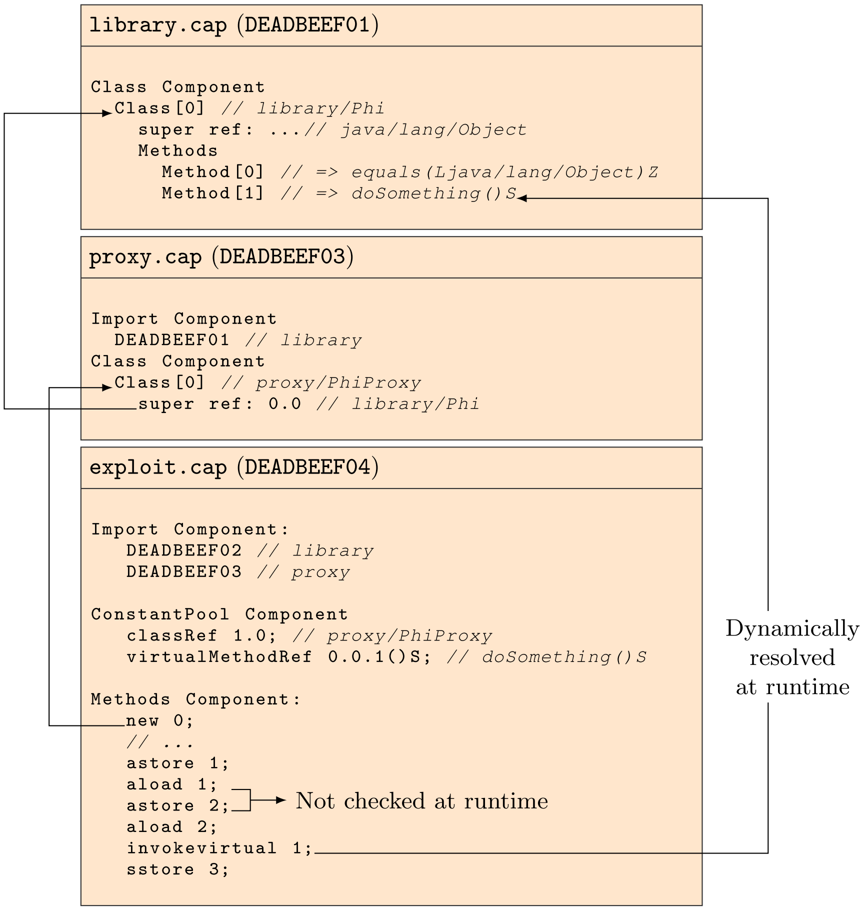


Fig. 3. BCV view when verifying `exploit.cap` file.

During the verification, the BCV performs its checks based on `export` files content:

– `proxy export` file states that `PhiProxy` inherits from a class called `library.Phi`. The missing information here is that this class must come from `library` package with `DEADBEEF01` AID.
– `library export` file with `DEADBEEF02` AID states that it contains a class named `library.Phi`. When verifying `exploit` package, the BCV only considers this `library` package based on the `Import` component of `exploit`.

On the other hand, at runtime, the JCVM tries to resolve the `doSomething` virtual method upon the `invokevirtual 1` instruction. To do this, the class hierarchy is browsed until finding the method token. Due to the similar construction, the `doSomething()` methods of both `library` packages have the same method token value.



`library.cap` (DEADBEEF01)

```
Class Component
  Class[0] // library/Phi
    super ref: ...// java/lang/Object
    Methods
      Method[0] // => equals(Ljava/lang/Object)Z
      Method[1] // => doSomething()S
```

`proxy.cap` (DEADBEEF03)

```
Import Component
  DEADBEEF01 // library
Class Component
  Class[0] // proxy/PhiProxy
    super ref: 0.0 // library/Phi
```

`exploit.cap` (DEADBEEF04)

```
Import Component:
   DEADBEEF02 // library
   DEADBEEF03 // proxy

ConstantPool Component
   classRef 1.0; // proxy/PhiProxy
   virtualMethodRef 0.0.1()S; // doSomething()S

Methods Component:
   new 0;
   // ...
   astore 1;
   aload 1;
   astore 2;
   aload 2;
   invokevirtual 1;
   sstore 3;
```

Fig. 4. Runtime view when executing `exploit.cap`.

From the JCVM point of view, Fig. 4, the actual class hierarchy of the currently accessed `Object` class:

`PhiProxy` → `Phi` (from `library` package with `DEADBEEF01` AID) → `Object`.

Therefore, when interpreting the `invokevirtual` instruction, the found method is the one from `library` package with `DEADBEEF01` AID: this method returns nothing (`void` type).

In line 10 in Listing 1.8, when returning from `doSomething()` method, a value is expected from the stack to store it in variable called `result`. During runtime, as the called method returns nothing the value is popped from an empty stack: a stack underflow is obtained.

This whole construction is allowed by the BCV because of the asymmetry in the import mechanism described in Sect. 2. In this Section, a stack underflow is demonstrated as example but various kinds of exploitation are described in Sect. 3.3.

3.3 Variations and Exploitation of Such an Attack

We have seen in Sect. 3.2 that a stack underflow attack can be performed using a specific construction that induces errors in the BCV import resolution. Using the same principle, a stack overflow attack can also be performed, by switching the two `library` packages.

The same principle can also be applied on the Java Card standard API. For instance, the attacker can create its own `javacardx.crypto` package with its own `Cipher` class (containing for instance methods with a different signature than expected). Using a `proxy` package in which a class inherits from the attacker's `Cipher` class, the principle described in Sect. 3.2 applies.

A type confusion attack can also be performed by replacing for instance Listings 1.5 and 1.6 by Listings 1.9 and 1.10. Indeed the `confusion()` method from package with `DEADBEEF01` AID will be called instead of the other one, transforming the `short` argument in a reference type.

Listing 1.9. `library` package with `DEADBEEF01` AID for type confusion.

```
package library;

public class Phi {
  public Object confusion(
      Object o){
    return o;
} }
```

Listing 1.10. `library` package with `DEADBEEF02` AID for type confusion.

```
package library;

public class Phi {
  public Object confusion(
      short s) {
    return null;
} }
```

This can even be performed with the `Object` class itself, in `java.lang` package. This allows to redefine a complete class hierarchy (with `Exception` and all the Java Card standard API). However, it must be noted that defining `Object` class in a CAP file leads to set very specific values in some structures of the CAP. For instance, the `super_class_ref` field of `class_info` structure in `Class` component has value `0xFFFF`. This value induces errors during CAP file loading on many public Java Card platforms. These errors suggest that the loader of such products is not designed to load a new Java Card class hierarchy root.

In Sect. 3.1, the two `Phi` classes have the same structure: they both inherit from `Object` and they both have the same number of public methods. However, if the number of public methods is different, calling a method in the `exploit` package may result in calling an actually non-existing method. Depending on the JCVM implementation, runtime may have several reactions, but overflow in the `public_virtual_method_table` is very likely to happen.

However, the `Phi` attack principle is not a full attack path by itself. Indeed the obtained overflow/underflow must still be exploited on a targeted device with a specific payload. Many state-of-the-art attacks [1,3–5,8–10,12,17] are detected by the BCV. Combined with `Phi` attack principle, these attacks become full exploitations that disclose sensitive assets without being detected by the BCV.

4 Discussion on Countermeasures

Our contribution was performed on Java Card specifications 3.0.5. However, when packages described in Sect. 3.1 are checked by the BCV provided by the Java Card 3.1.0 toolchain, the following log is obtained:

Listing 1.11. BCV log on `proxy` package

```
INFOS: [v3.1.0] Off-Card Verifier
INFOS: Export file library\javacard\library.exp is in an older export
    file format. Please update the export file to format 2.3.
INFOS: Export file proxy\javacard\proxy.exp is in an older export file
    format. Please update the export file to format 2.3.
INFOS: Verifying CAP file proxy\javacard\proxy.cap
INFOS: Verification completed with 0 warnings and 0 errors.
```

As stated in Sect. 3.1, the BCV raises no warning and no error, validating the CAP and `export` files. However, information about `export` files version is returned.

Indeed only `export` files in version `2.2`, specified in [14], have been used. Version `2.3` is described in [16]. Nevertheless, `export` files in version `2.2` are still accepted by the BCV 3.1.0 as valid format, with only an information indicating that a new format is available.

Version `2.3` of `export` file format adds supplementary information in the file. Among modifications, a new structure is added containing the AID value, the minor and major version of each package referenced in the `export` file. The AID value is the missing information that prevented the BCV to detect the attack attempted in Sect. 3.1. Therefore, when `export` files are generated in version `2.3`, the construction shown in Sect. 3.1 is successfully detected by the BCV as malformed.

Before loading one or several CAP files in a Java-Card based product, the latest version of the BCV must be executed in order to ensure that the loaded code is not malicious. However, more than just running the BCV, the entity performing the verification should also check that `export` files provided are in version `2.3`. Ensuring the version is `2.3` allows to detect potential malicious applications to be loaded.

5 Conclusion

We show in this article how a missing information in the `export` file allows an attacker to abuse the BCV checks during packages import resolution. This could lead an attacker to execute malicious pieces of code within a verified application allowing to potentially break the Java Card security model.

This kind of issue can be countered by denying the use of `export` file format older than `2.3` even if the latest BCV version still accepts `export` files in `2.2` version.

The identification of this missing information in `export` files allowing to attack Java Card products opens perspective and potential future work on finding other kind of information that would be completely or partially missing.

Following our responsible disclosure policy, as far as we know, all the Java Card platform developers concerned by this vulnerability were informed before the publication of this paper.

Acknowledgments. A very special thanks to my wife, Marie-Philomène Dubreuil, who accompanied me during all these hours of work on this research topic. This attack is named after her.

Jean Dubreuil

References

1. Bouffard, G., Iguchi-Cartigny, J., Lanet, J.-L.: Combined software and hardware attacks on the Java card control flow. In: Prouff, E. (ed.) CARDIS 2011. LNCS, vol. 7079, pp. 283–296. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-27257-8_18
2. Bouffard, G., Khefif, T., Lanet, J., Kane, I., Salvia, S.C.: Accessing secure information using export file fraudulence. In: Crispo, B., Sandhu, R.S., Cuppens-Boulahia, N., Conti, M., Lanet, J. (eds.) 2013 International Conference on Risks and Security of Internet and Systems (CRiSIS), La Rochelle, France, 23–25 October 2013, pp. 1–5. IEEE (2013). https://doi.org/10.1109/CRiSIS.2013.6766346
3. Bouffard, G., Lanet, J.-L.: Reversing the operating system of a Java based smart card. J. Comput. Virol. Hacking Tech. **10**(4), 239–253 (2014). https://doi.org/10.1007/s11416-014-0218-7
4. Bouffard, G., Lanet, J.: The ultimate control flow transfer in a Java based smart card. Comput. Secur. **50**, 33–46 (2015). https://doi.org/10.1016/j.cose.2015.01.004
5. Faugeron, E.: Manipulating the frame information with an underflow attack. In: Francillon, A., Rohatgi, P. (eds.) CARDIS 2013. LNCS, vol. 8419, pp. 140–151. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08302-5_10
6. Faugeron, E., Valette, S.: How to hoax an off-card verifier. e-smart (2010)
7. GlobalPlatform: Card Specification. GlobalPlatform Inc., 2.2.1 edn. (January 2011)
8. Hamadouche, S., et al.: Subverting byte code linker service to characterize Java card API. In: 7th Conference on Network and Information Systems Security (SAR-SSI), 22–25 May 2012, pp. 75–81 (2012)
9. Hamadouche, S., Lanet, J.: Virus in a smart card: myth or reality? J. Inf. Secur. Appl. **18**(2–3), 130–137 (2013). https://doi.org/10.1016/j.jisa.2013.08.005
10. Lancia, J.: Java card combined attacks with localization-agnostic fault injection. In: Mangard, S. (ed.) CARDIS 2012. LNCS, vol. 7771, pp. 31–45. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37288-9_3
11. Lancia, J., Bouffard, G.: Java card virtual machine compromising from a bytecode verified applet. In: Homma, N., Medwed, M. (eds.) CARDIS 2015. LNCS, vol. 9514, pp. 75–88. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-31271-2_5

12. Mostowski, W., Poll, E.: Malicious code on Java card smartcards: attacks and countermeasures. In: Grimaud, G., Standaert, F.-X. (eds.) CARDIS 2008. LNCS, vol. 5189, pp. 1–16. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85893-5_1
13. Oracle: Java Card Technology - Providing a secure and ubiquitous platform for smart cards. Technical report, Oracle, Security Evaluations, Oracle Corporation, 500 Oracle Parkway, Redwood Shores, CA 94065 (2012). www.oracle.com/technetwork/java/embedded/javacard/documentation/datasheet-149940.pdf
14. 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)
15. Oracle: Java card system - open configuration protection profile. Protection Profile versoin 3.0.5, Oracle, Security Evaluations, Oracle Corporation, 500 Oracle Parkway, Redwood Shores, CA 94065 (December 2017)
16. Oracle: Java Card 3 Platform, Virtual Machine Specification, Classic Edition. No. Version 3.1, Oracle, Oracle America Inc., 500 Oracle Parkway, Redwood City, CA 94065 (February 2021)
17. Razafindralambo, T., Bouffard, G., Lanet, J.-L.: A friendly framework for hidding *fault enabled virus* for Java based smartcard. In: Cuppens-Boulahia, N., Cuppens, F., Garcia-Alfaro, J. (eds.) DBSec 2012. LNCS, vol. 7371, pp. 122–128. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31540-4_10