Risk mitigation for sensitive applets in a multi-application context

Marc Witteman, CTO Riscure Security Lab

witteman@riscure.com

Abstract: Modern Java Card platforms support co-existence of multiple applets and post-issuance applet loading. Owners of sensitive applets seek assurance that their assets are secure in the presence of other applets. We explain security guidelines proposed by Global Platform and introduce a new Riscure product to automatically verify basic applet security, and support additional in-depth analysis.

Introduction

Java Cards typically host multiple applets. These are provided in binary CAP files, containing library code or applets. The CAP files may or may not be security sensitive. Typically owners of sensitive applications (e.g. payment applets), would like assurance that other applications, referred to as **basic applets**, are not harmful.

While a full evaluation of all applets on a card would be technically possible, there are practical hurdles. This is because the java card vendor, who seeks certification of the product for payment purposes, may not have access to all code an issuer may wish to load. A reason for this could be because the issuer gets code through alternative channels (application providers), or because the code is not yet available and might be added later in the product life cycle (post-issuance). On top of that, application providers are not eager to provide source code of their product in order to protect their IP.

An alternative approach would be to have a (partially) automated verification of binary code that may even be done outside the certification process and would not necessarily require expert level security knowledge. This could help payment schemes and other owners of sensitive applications to gain confidence that their assets are secure. This document explains how Riscure can support this approach and help mitigate risk.

Threats and attacks

Although java card incorporates security by design, there are several threats that basic applets can pose to sensitive applets:

1. Resource exhaustion, e.g.
   - Unavailability of memory or CPU time
   - Card blocked or terminated
2. Exposure of sensitive shared resources, e.g.
   - PIN unblock, reset counter, change (may allow PIN brute forcing or DoS)
   - Cryptographic implementation (may allow unlimited side channel profiling)

3. Virtual Machine Integrity breach (can compromise all assets), due to e.g.
    - corrupt (incompatible or ill-formed) binary code
    - fault injection

Part of these threats can be addressed in the regular certification process. E.g. the security lab could verify that the product does not leak secret keys, even when exposed to a brute force attack. Also the security lab could challenge the robustness of the virtual machine and measure its resistance against fault injection.

Resource exhaustion is bad, but should be detected during functional testing that both issuers and application providers are expected to do. This threat should not be the primary concern for the owners of sensitive applications.

It is much harder to prevent at issuance time that post-issuance basic applications will not expose the global PIN code, or include corrupt code. These threats could be addressed with a new approach.

Requirements

Global Platform is a standards body well known for defining card management standards. Recently they defined guidelines for the verification of Basic Applets. These are available as “GlobalPlatform Card Composition Model Security Guidelines for Basic Applications v2.0”, on web page http://www.globalplatform.org/specificationscard.asp.

The guidelines focus on the interaction between library packages and applet packages. The former package exports methods (functions), which are used by the latter.

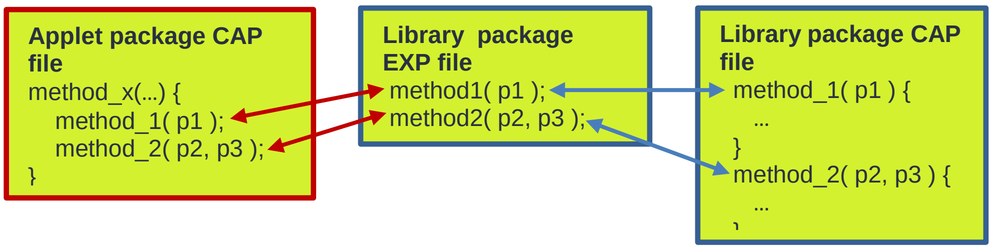


Figure 1: Interaction between applet and library packages

Figure 1 shows the interaction between these packages. The library package implements methods in its CAP file, these are advertised in the corresponding EXP file. The applet package calls methods in the CAP file conform their definition in the EXP file. Note that the CAP files are loaded on the card, while the EXP files are kept off-card as a way to link applet packages to library packages.

As the library EXP file is the link between an applet and library package it is important that this EXP file is compatible with both library and applet CAP files. This compatibity means that parameter and return types in applet and library package match.

Incompatibilities between packages can breach security. It may provide a basic applet access to restricted memory and compromize the security of all assets. More information on these attacks can be found in https://www.riscure.com/archive/ISB0808MW.pdf and http://wwwhome.ewi.utwente.nl/~mostowskiwi/papers/cardis2008.pdf.

The compatibility of packages can be verified automatically, which is mandated in the guidelines. Specifically, the GP document includes four requirements:

1. **Library compatibility**
   Libraries on the card shall be compatible with libraries assumed by the Basic Application;
2. **Version update policy**
   Libraries changes shall always increase the minor version, and shall increase the major version if a method signature changes;
3. **Restricted access to sensitive shared resources**
   Basic Application shall not unblock, reset counter, or change the Global Platform CVM (PIN);
4. **Verification**
   Basic application must pass latest byte code verifier.

Payment schemes support these guidelines, but may also desire additional effort by reviewing the code and check for potentially dangerous behavior.

Verification environment

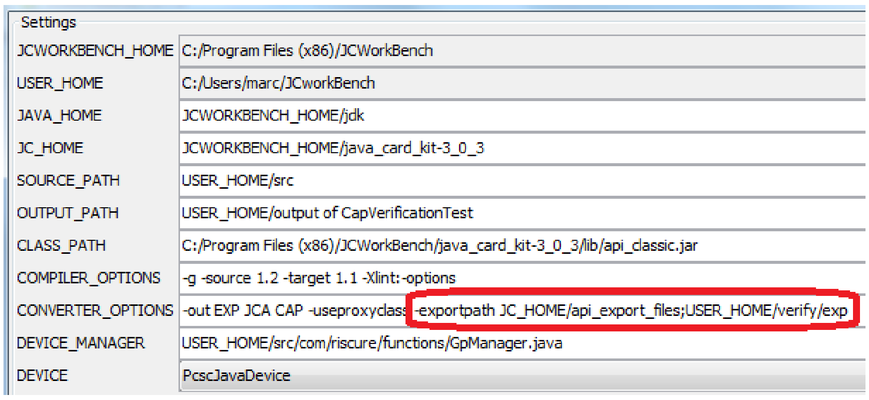
Riscure provides in version 2.8 of her JCworkBench tool a solution to verify basic applets. In this section we detail how a verification session is prepared and started.

Configuration

JCworkBench is a software platform running on a PC that provides multiple tools to work with, and analyze the security of, Java Card code. To analyze the security of a CAP file an analyst would start by copying a CAP file along with all export files of the pre-existing code it depends on to the storage space accessible by the computer that JCworkBench runs on.

For the verification it is important that the related export files are found, but even stronger that only the correct export files are found. If several variants exist, it is essential that the export file corresponding to the actual code loaded on the card is provided for the verification. Therefore the user will configure locations (folders) where export files are to be found. At run time the verifier will look into those locations, the CAP file location, and all nested folders.

Figure 2 shows how the export file locations are configured in the convertor options. This allows all tools handling CAP files to find the right EXP files. In the example in the figure there are two folders indicated where export files are located for respectively the Java Card system and the user libraries. Note that symbolic links are used in the configuration dialogue to simplify referencing.



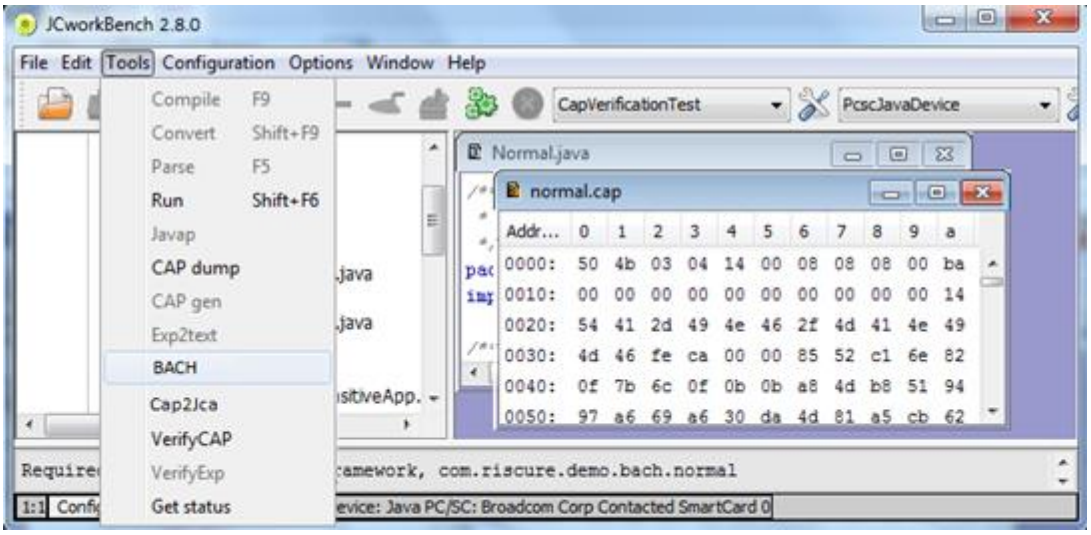
Settings

JCWORKBENCH_HOME	C:/Program Files (x86)/JCWorkBench
USER_HOME	C:/Users/marc/JCworkBench
JAVA_HOME	JCWORKBENCH_HOME/jdk
JC_HOME	JCWORKBENCH_HOME/java_card_kit-3_0_3
SOURCE_PATH	USER_HOME/src
OUTPUT_PATH	USER_HOME/output of CapVerificationTest
CLASS_PATH	C:/Program Files (x86)/JCWorkBench/java_card_kit-3_0_3/lib/api_classic.jar
COMPILER_OPTIONS	-g -source 1.2 -target 1.1 -Xlint:-options
CONVERTER_OPTIONS	-out EXP JCA CAP -useproxyclass -exportpath JC_HOME/api_export_files;USER_HOME/verify/exp
DEVICE_MANAGER	USER_HOME/src/com/riscure/functions/GpManager.java
DEVICE	PcscJavaDevice

Figure 2: Selecting export files for verification

Invocation

JCworkBench includes a series of applet verification tests invoked by selecting BACH (Basic Applet Check Help) under the Tools menu. BACH can run on an individual CAP file or on a set of CAP files. Individual CAP files are verified easiest by selecting the CAP file window (after opening the file), and then invoking BACH. This is shown in Figure 3.



```
Addr... 0  1  2  3  4  5  6  7  8  9  a
0000:   50 4b 03 04 14 00 08 08 08 00 ba
0010:   00 00 00 00 00 00 00 00 00 00 14
0020:   54 41 2d 49 4e 46 2f 4d 41 4e 49
0030:   4d 46 fe ca 00 00 85 52 c1 6e 82
0040:   0f 7b 6c 0f 0b 0b a8 4d b8 51 94
0050:   97 a6 69 a6 30 da 4d 81 a5 cb 62
```

```
Require… …amework, com.riscure.demo.bach.normal
1:1 Confi… …evice: Java PC/SC: Broadcom Corp Contacted SmartCard 0
```

Figure 3: Invoking BACH on a single CAP file

Alternatively, BACH can be invoked when no CAP window is selected. In that case a dialog opens that allows for selecting an individual file, or a folder, that will be recursively scanned for CAP files to verify. This option is useful when a large collection of files is to be verified.

Identification

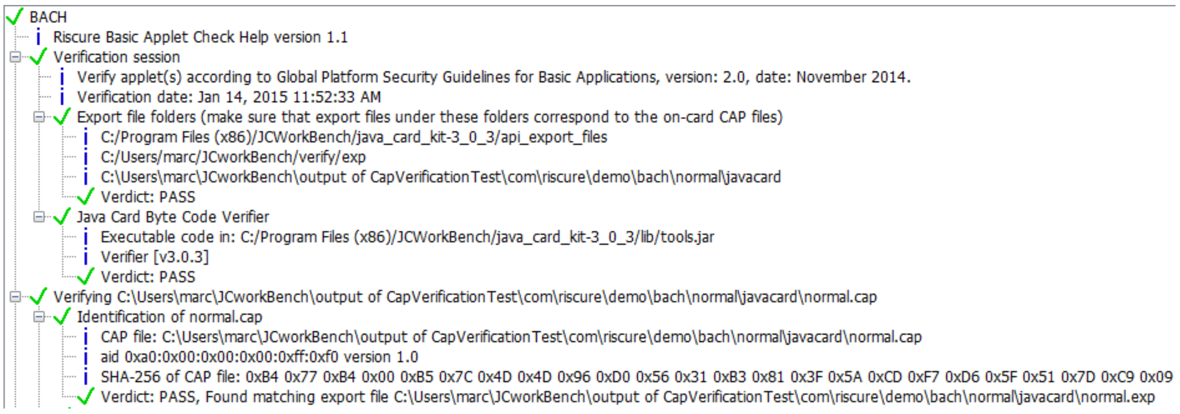
After invocation BACH will first report general information about the session and the CAP file. This information is helpful when sharing reports and understanding the context. It also facilitates reproducibility of test results. The following information is provided about the session:

- BACH version and Global Platform guidelines version
- Verification date
- Export file folders
- Java Card Byte Code Verifier

The CAP file is identified with the following information:

- CAP file name and location
- AID and version number
- SHA-256 hash over the binary contents of the file
- Corresponding EXP file name and location

Figure 4 shows how this information is presented.



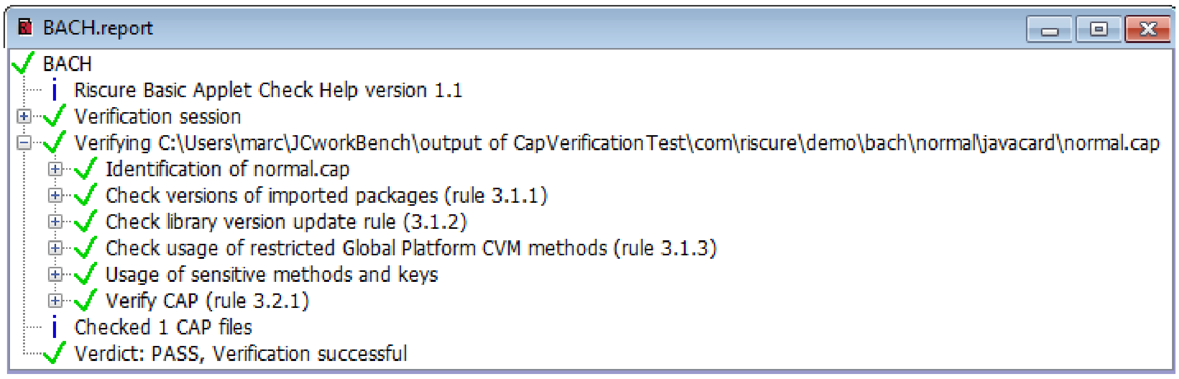
```
✓ BACH
    i Riscure Basic Applet Check Help version 1.1
  ✓ Verification session
      i Verify applet(s) according to Global Platform Security Guidelines for Basic Applications, version: 2.0, date: November 2014.
      i Verification date: Jan 14, 2015 11:52:33 AM
    ✓ Export file folders (make sure that export files under these folders correspond to the on-card CAP files)
        i C:/Program Files (x86)/JCWorkBench/java_card_kit-3_0_3/api_export_files
        i C:/Users/marc/JCworkBench/verify/exp
        i C:\Users\marc\JCworkBench\output of CapVerificationTest\com\riscure\demo\bach\normal\javacard
      ✓ Verdict: PASS
    ✓ Java Card Byte Code Verifier
        i Executable code in: C:/Program Files (x86)/JCWorkBench/java_card_kit-3_0_3/lib/tools.jar
        i Verifier [v3.0.3]
      ✓ Verdict: PASS
  ✓ Verifying C:\Users\marc\JCworkBench\output of CapVerificationTest\com\riscure\demo\bach\normal\javacard\normal.cap
    ✓ Identification of normal.cap
        i CAP file: C:\Users\marc\JCworkBench\output of CapVerificationTest\com\riscure\demo\bach\normal\javacard\normal.cap
        i aid 0xa0:0x00:0x00:0x00:0xff:0xf0 version 1.0
        i SHA-256 of CAP file: 0xB4 0x77 0xB4 0x00 0xB5 0x7C 0x4D 0x4D 0x96 0xD0 0x56 0x31 0xB3 0x81 0x3F 0x5A 0xCD 0xF7 0xD6 0x5F 0x51 0x7D 0xC9 0x09
      ✓ Verdict: PASS, Found matching export file C:\Users\marc\JCworkBench\output of CapVerificationTest\com\riscure\demo\bach\normal\javacard\normal.exp
```

Figure 4: identification report

Global Platform guidelines verification

In this section we detail on the verification process and reporting for the four GP (Global Platform) guidelines. This verification process does not require a high degree of security expertise, and is therefore optimized for clear and concise reporting. Each guideline is indicated separately and concluded with an unambiguous verdict.

Figure 5 shows the collapsed report including sections for each of the four global platform rules (referring to sections 3.1.1, 3.1.2, 3.1.3, and 3.2.1 in the GP guidelines). In this example the session is successful and all tests pass.



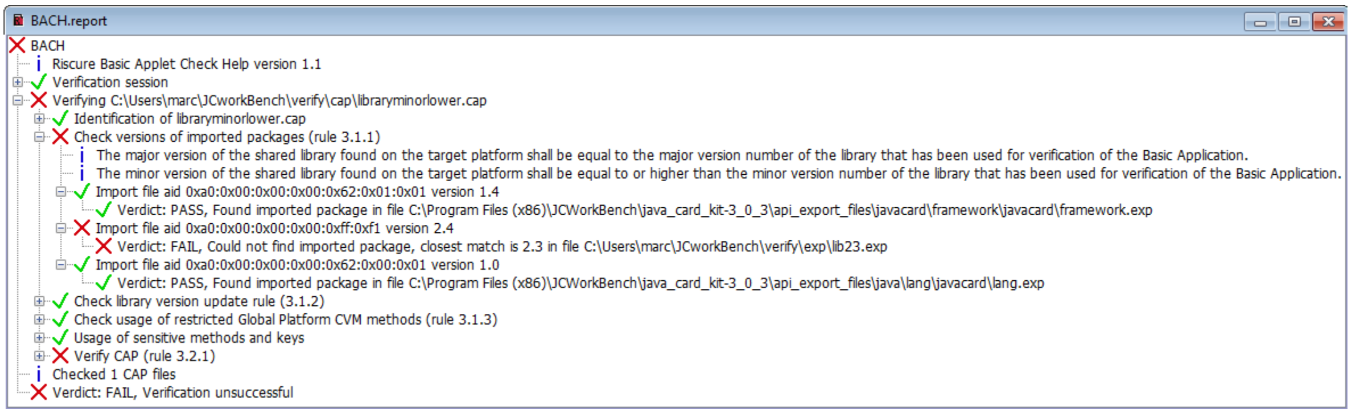
```
BACH.report
✓ BACH
   i Riscure Basic Applet Check Help version 1.1
 ⊞ ✓ Verification session
 ⊟ ✓ Verifying C:\Users\marc\JCworkBench\output of CapVerificationTest\com\riscure\demo\bach\normal\javacard\normal.cap
    ⊞ ✓ Identification of normal.cap
    ⊞ ✓ Check versions of imported packages (rule 3.1.1)
    ⊞ ✓ Check library version update rule (3.1.2)
    ⊞ ✓ Check usage of restricted Global Platform CVM methods (rule 3.1.3)
    ⊞ ✓ Usage of sensitive methods and keys
    ⊞ ✓ Verify CAP (rule 3.2.1)
   i Checked 1 CAP files
   ✓ Verdict: PASS, Verification successful
```

Figure 5: Collapsed BACH report

Library Compatibility

Applets may require code provided by libraries, these are CAP files that just include static methods that can be used by any applet. It is important that applets are developed and verified against compatible library versions, which are the same as those on the target smart card platform. In accordance with GP rule 3.1.1 it is tested that each package imported by an applet package matches its expected version, or - as an allowed deviation - has a minor version number that is higher than expected. In the latter case the actually imported library is newer and may have additional functionality, but should not have changed the signatures or behavior of the earlier version.

BACH will list for every imported file the expected version number, and then first search for an exact version match, and if unsuccessful search for a newer version. Figure 6 shows an example where the best found export file (lib23.exp) has a lower version (2.3) than expected (2.4). This violates rule 3.1.1, and results in a verification failure. Note that the same problem is also detected by the Java Card Byte Code Verifier (rule 3.2.1).

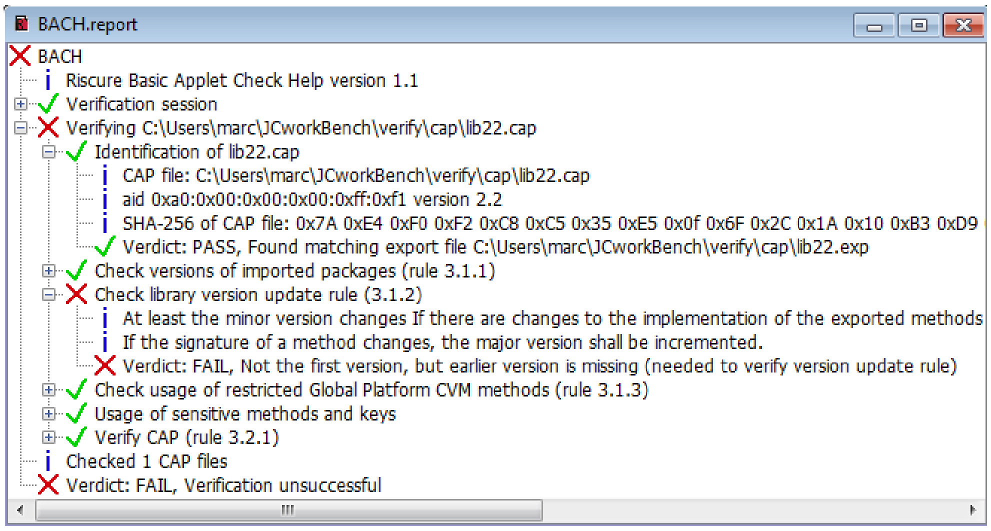


```
BACH.report
✗ BACH
  i Riscure Basic Applet Check Help version 1.1
  ✓ Verification session
  ✗ Verifying C:\Users\marc\JCworkBench\verify\cap\libraryminorlower.cap
    ✓ Identification of libraryminorlower.cap
    ✗ Check versions of imported packages (rule 3.1.1)
      i The major version of the shared library found on the target platform shall be equal to the major version number of the library that has been used for verification of the Basic Application.
      i The minor version of the shared library found on the target platform shall be equal to or higher than the minor version number of the library that has been used for verification of the Basic Application.
      ✓ Import file aid 0xa0:0x00:0x00:0x00:0x62:0x01:0x01 version 1.4
        ✓ Verdict: PASS, Found imported package in file C:\Program Files (x86)\JCWorkBench\java_card_kit-3_0_3\api_export_files\javacard\framework\javacard\framework.exp
      ✗ Import file aid 0xa0:0x00:0x00:0x00:0xff:0xf1 version 2.4
        ✗ Verdict: FAIL, Could not find imported package, closest match is 2.3 in file C:\Users\marc\JCworkBench\verify\exp\lib23.exp
      ✓ Import file aid 0xa0:0x00:0x00:0x00:0x62:0x00:0x01 version 1.0
        ✓ Verdict: PASS, Found imported package in file C:\Program Files (x86)\JCWorkBench\java_card_kit-3_0_3\api_export_files\java\lang\javacard\lang.exp
    ✓ Check library version update rule (3.1.2)
    ✓ Check usage of restricted Global Platform CVM methods (rule 3.1.3)
    ✓ Usage of sensitive methods and keys
    ✗ Verify CAP (rule 3.2.1)
  i Checked 1 CAP files
  ✗ Verdict: FAIL, Verification unsuccessful
```

Figure 6: Version check of imported packages

Version update policy

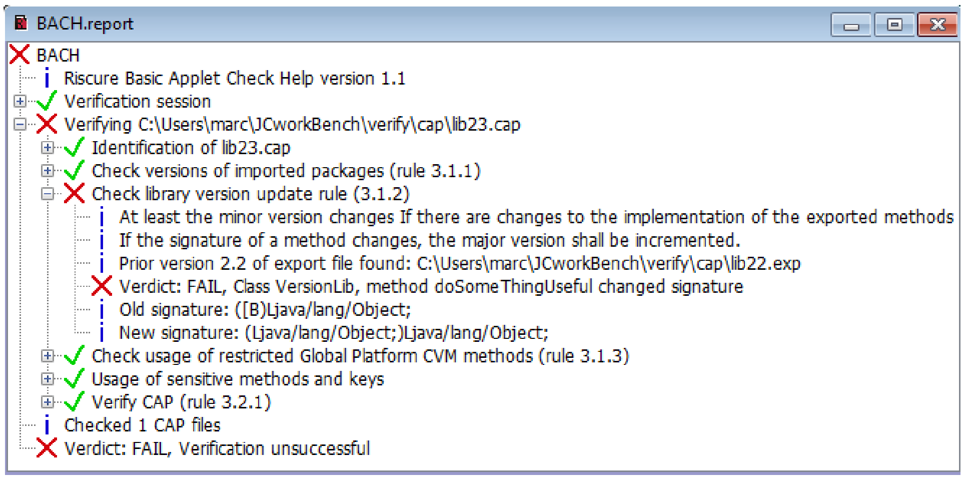
Libraries that are updated shall also change their version number. If a library changes the signature of a method it should increase the major version number. To evaluate this rule the user shall have the export files of earlier versions of a library available for comparison. Figure 7 shows an example where the evaluated library version (2.2) is clearly not the first version, but an export file for a prior version is missing. In absence of the prior version the verification fails as it cannot be confirmed that the version update conforms to the guideline.



```
BACH.report
✗ BACH
  i Riscure Basic Applet Check Help version 1.1
  ✓ Verification session
  ✗ Verifying C:\Users\marc\JCworkBench\verify\cap\lib22.cap
    ✓ Identification of lib22.cap
      i CAP file: C:\Users\marc\JCworkBench\verify\cap\lib22.cap
      i aid 0xa0:0x00:0x00:0x00:0xff:0xf1 version 2.2
      i SHA-256 of CAP file: 0x7A 0xE4 0xF0 0xF2 0xC8 0xC5 0x35 0xE5 0x0f 0x6F 0x2C 0x1A 0x10 0xB3 0xD9
      ✓ Verdict: PASS, Found matching export file C:\Users\marc\JCworkBench\verify\cap\lib22.exp
    ✓ Check versions of imported packages (rule 3.1.1)
    ✗ Check library version update rule (3.1.2)
      i At least the minor version changes If there are changes to the implementation of the exported methods
      i If the signature of a method changes, the major version shall be incremented.
      ✗ Verdict: FAIL, Not the first version, but earlier version is missing (needed to verify version update rule)
    ✓ Check usage of restricted Global Platform CVM methods (rule 3.1.3)
    ✓ Usage of sensitive methods and keys
    ✓ Verify CAP (rule 3.2.1)
  i Checked 1 CAP files
  ✗ Verdict: FAIL, Verification unsuccessful
```

Figure 7: Library missing prior export file

Another example is given in Figure 8, where a newer library version has changed a method signature without increasing the major version number. This is a risk as this an applet may abuse such library to mount a type confusion attack leading to full asset compromise.

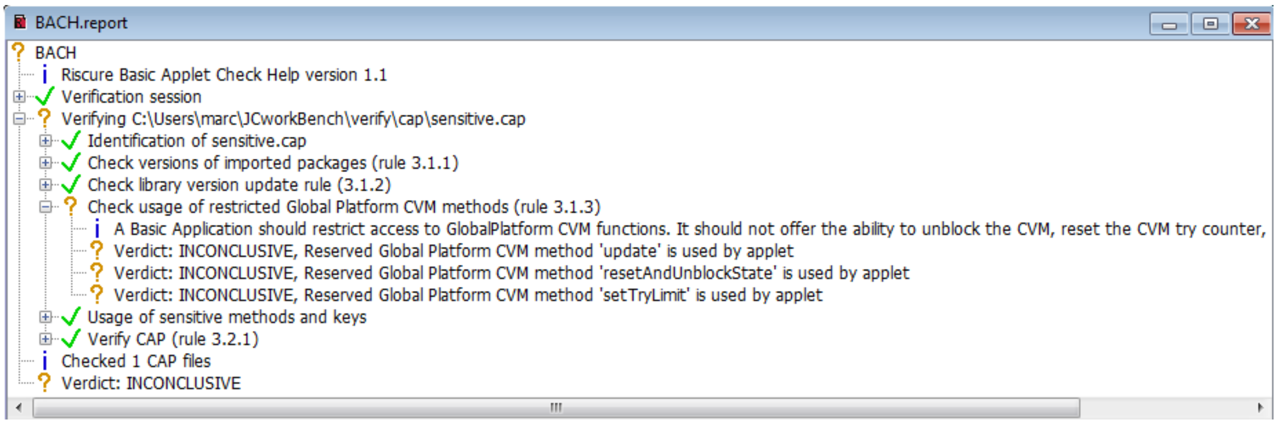


```
BACH.report
BACH
  Riscure Basic Applet Check Help version 1.1
  Verification session
  Verifying C:\Users\marc\JCworkBench\verify\cap\lib23.cap
    Identification of lib23.cap
    Check versions of imported packages (rule 3.1.1)
    Check library version update rule (3.1.2)
      At least the minor version changes If there are changes to the implementation of the exported methods
      If the signature of a method changes, the major version shall be incremented.
      Prior version 2.2 of export file found: C:\Users\marc\JCworkBench\verify\cap\lib22.exp
      Verdict: FAIL, Class VersionLib, method doSomeThingUseful changed signature
      Old signature: ([B)Ljava/lang/Object;
      New signature: (Ljava/lang/Object;)Ljava/lang/Object;
    Check usage of restricted Global Platform CVM methods (rule 3.1.3)
    Usage of sensitive methods and keys
    Verify CAP (rule 3.2.1)
  Checked 1 CAP files
  Verdict: FAIL, Verification unsuccessful
```

Figure 8: Functional library update fails to increase major version

Restricted access to sensitive shared resources

GP provides a library package including a CVM (Cardholder Verification Method) concept allowing applets to share a PIN code. The CVM includes methods to change the value or status of the PIN code. Applets that invoke these methods can expose the CVM and threaten other applets that rely on them. Figure 9 shows a verification session that detects the use of these methods. Although there is a potential danger, this is not necessarily a violation as the use of these methods might still be legitimate and secure if sufficient authentication is applied. Therefore the test does not return a Fail, but an Inconclusive verdict. The user may then decide whether additional analysis is needed.



```
BACH.report
BACH
  Riscure Basic Applet Check Help version 1.1
  Verification session
  Verifying C:\Users\marc\JCworkBench\verify\cap\sensitive.cap
    Identification of sensitive.cap
    Check versions of imported packages (rule 3.1.1)
    Check library version update rule (3.1.2)
    Check usage of restricted Global Platform CVM methods (rule 3.1.3)
      A Basic Application should restrict access to GlobalPlatform CVM functions. It should not offer the ability to unblock the CVM, reset the CVM try counter,
      Verdict: INCONCLUSIVE, Reserved Global Platform CVM method 'update' is used by applet
      Verdict: INCONCLUSIVE, Reserved Global Platform CVM method 'resetAndUnblockState' is used by applet
      Verdict: INCONCLUSIVE, Reserved Global Platform CVM method 'setTryLimit' is used by applet
    Usage of sensitive methods and keys
    Verify CAP (rule 3.2.1)
  Checked 1 CAP files
  Verdict: INCONCLUSIVE
```

Figure 9: Applet uses GP CVM methods

Byte Code Verification

In addition to the topics described in the sections above there are many more possible issues in CAP files. These relate to the coding problems within the CAP file. The Java Card Byte Code Verifier (BCV) should detect these problems. The best way to detect potential issues is to run the BCV. Figure 10 shows an example where a corrupted CAP file is detected by the BCV. JCworkBench automatically links the verification process with the external BCV tool, and copies the result in the test report. Bach also reports the version number of the BCV allowing the user to verify that the newest BCV is used. Note that newer BCV versions can be obtained through the Oracle web site:
http://www.oracle.com/technetwork/java/embedded/javacard/downloads/javacard-sdk-2043229.html

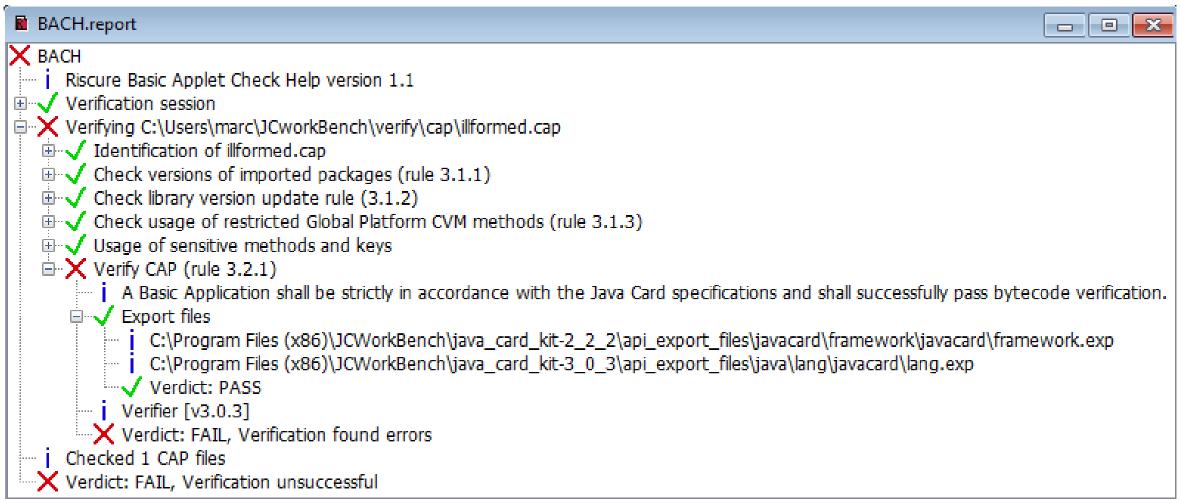

```
BACH.report
✗ BACH
    i Riscure Basic Applet Check Help version 1.1
  ⊞ ✓ Verification session
  ⊟ ✗ Verifying C:\Users\marc\JCworkBench\verify\cap\illformed.cap
      ⊞ ✓ Identification of illformed.cap
      ⊞ ✓ Check versions of imported packages (rule 3.1.1)
      ⊞ ✓ Check library version update rule (3.1.2)
      ⊞ ✓ Check usage of restricted Global Platform CVM methods (rule 3.1.3)
      ⊞ ✓ Usage of sensitive methods and keys
      ⊟ ✗ Verify CAP (rule 3.2.1)
            i A Basic Application shall be strictly in accordance with the Java Card specifications and shall successfully pass bytecode verification.
          ⊟ ✓ Export files
                i C:\Program Files (x86)\JCWorkBench\java_card_kit-2_2_2\api_export_files\javacard\framework\javacard\framework.exp
                i C:\Program Files (x86)\JCWorkBench\java_card_kit-3_0_3\api_export_files\java\lang\javacard\lang.exp
                ✓ Verdict: PASS
            i Verifier [v3.0.3]
            ✗ Verdict: FAIL, Verification found errors
    i Checked 1 CAP files
    ✗ Verdict: FAIL, Verification unsuccessful
```

Figure 10: Byte Code Verification problem detected

Reverse engineering

While the BACH tool implements all the checks proposed by Global Platform, it is sometimes desirable to perform additional analysis. For instance this may occur when usage of sensitive functionality is detected, or when more in-depth analysis is expected in the context of certification.

JCworkBench supports listing of sensitive functionality usage and CAP file reverse engineering.

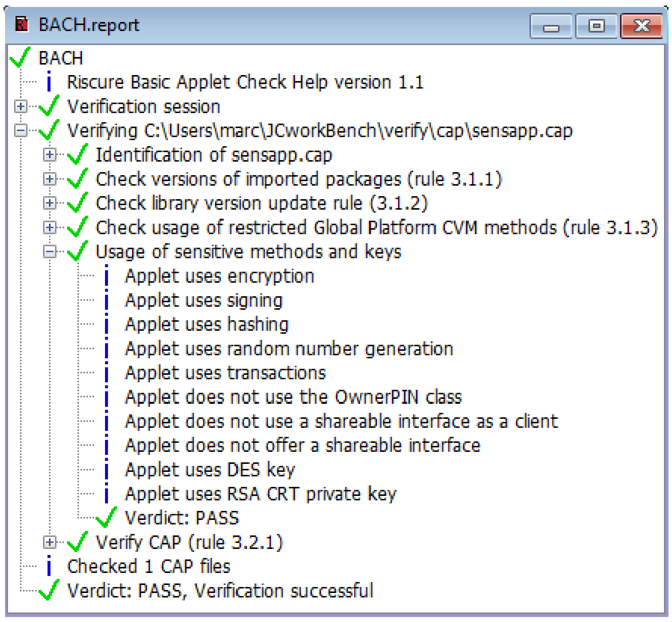
Sensitive functionality usage

The BACH tool inspects a CAP file for usage of the following functionality:

- Encryption and signing
- Random number generation
- Hashing

- Transactions
- Firewall interaction between applets
- PIN
- Secret or private key types used.

This sensitive functionality is typically used in applets that may be vulnerable to, or involved in attacks. The identification of sensitive functionality allows the user to decide on the need of further exploration. Figure 11 shows how BACH reports the usage of sensitive functionality.



```
✓ BACH
   i Riscure Basic Applet Check Help version 1.1
 ⊞ ✓ Verification session
 ⊟ ✓ Verifying C:\Users\marc\JCworkBench\verify\cap\sensapp.cap
    ⊞ ✓ Identification of sensapp.cap
    ⊞ ✓ Check versions of imported packages (rule 3.1.1)
    ⊞ ✓ Check library version update rule (3.1.2)
    ⊞ ✓ Check usage of restricted Global Platform CVM methods (rule 3.1.3)
    ⊟ ✓ Usage of sensitive methods and keys
          i Applet uses encryption
          i Applet uses signing
          i Applet uses hashing
          i Applet uses random number generation
          i Applet uses transactions
          i Applet does not use the OwnerPIN class
          i Applet does not use a shareable interface as a client
          i Applet does not offer a shareable interface
          i Applet uses DES key
          i Applet uses RSA CRT private key
          ✓ Verdict: PASS
    ⊞ ✓ Verify CAP (rule 3.2.1)
   i Checked 1 CAP files
   ✓ Verdict: PASS, Verification successful
```

Figure 11: Sensitive method analysis

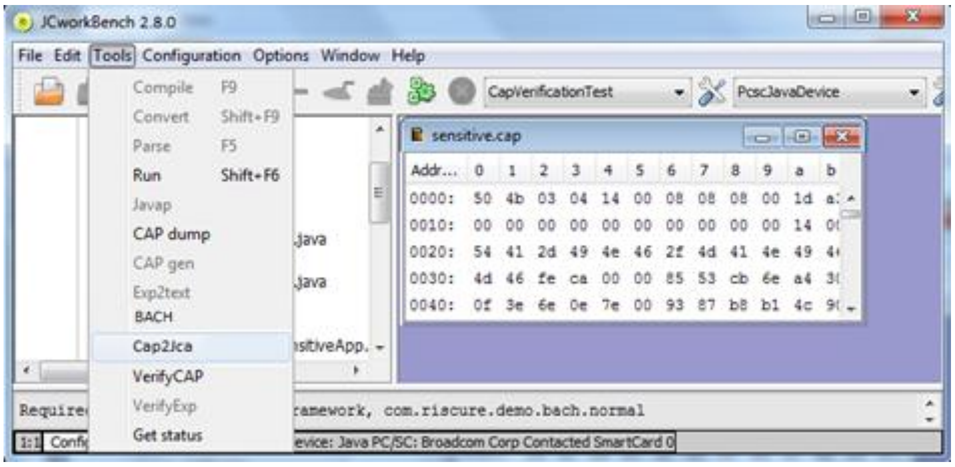
Reverse Engineering

JCworkBench includes a tool called Cap2Jca to convert a binary CAP file into human readable assembly code, the JCA file format. The JCA file format is an internal format used by the Java Card development kit tools. It is loosely based on "Jasmin" (http://jasmin.sourceforge.com).

The JCA files generated by Cap2jca can be converted back into CAP files by using the JC development kit tools. Cap2jca optionally and automatically applies symbol information such as field, method, class and package names present in export files and CAP file debug components to improve readability in the generated JCA.

Both Cap2jca and BACH functionality are built on the same Riscure developed library designed to interpret and modify CAP files within JCworkBench. This unique feature is specifically developed by Riscure for this purpose, and is not commonly available in Java Card development or test tools.

This reverse engineering step allows further inspection of the code. By invoking Cap2Jca under the tools menu the conversion is started, as shown in Figure 12.

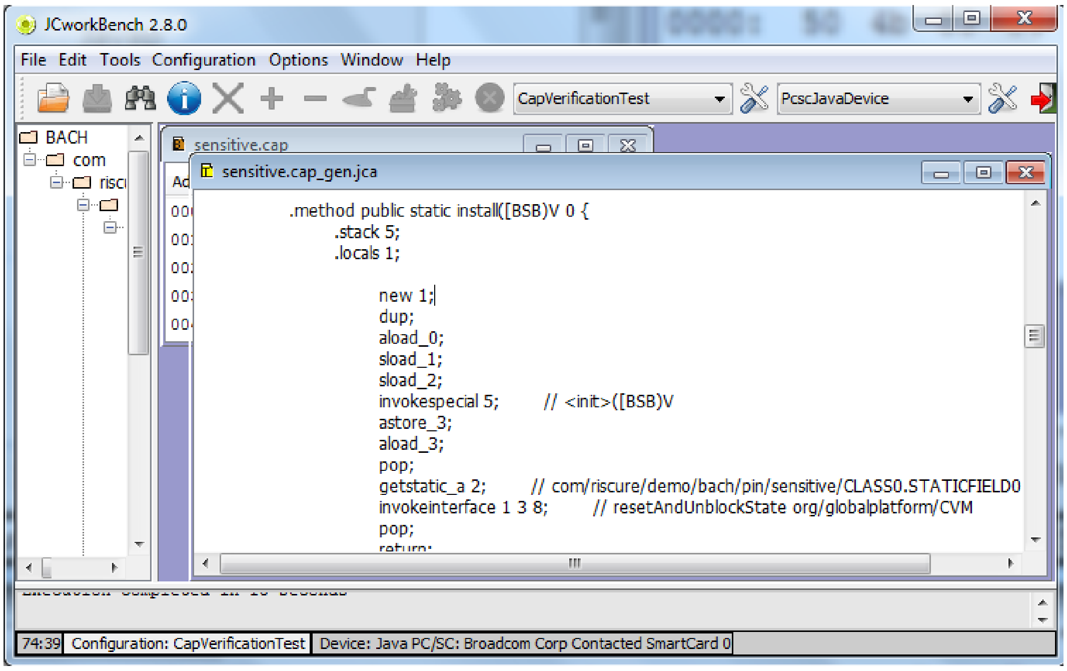


```
Addr... 0  1  2  3  4  5  6  7  8  9  a  b
0000:  50 4b 03 04 14 00 08 08 08 00 1d a
0010:  00 00 00 00 00 00 00 00 00 00 14 0
0020:  54 41 2d 49 4e 46 2f 4d 41 4e 49 4
0030:  4d 46 fe ca 00 00 85 53 cb 6e a4 3
0040:  0f 3e 6e 0e 7e 00 93 87 b8 b1 4c 9
```

```
Require  ramework, com.riscure.demo.bach.normal
1:1  Confi  evice: Java PC/SC: Broadcom Corp Contacted SmartCard 0
```

Figure 12: Invoking Cap2Jca

The resulting JCA file is shown in Figure 13. In this example additional inspection can be used to evaluate the use of the GP PIN unblock mechanism. The JCA code includes a reference in comments to the exact method name that is searched and detected. With a modest effort the user can verify here that the PIN unblock is only invoked at installation time, and that a brute force attack is not possible.



```
        .method public static install([BSB)V 0 {
                .stack 5;
                .locals 1;

                        new 1;
                        dup;
                        aload_0;
                        sload_1;
                        sload_2;
                        invokespecial 5;        // <init>([BSB)V
                        astore_3;
                        aload_3;
                        pop;
                        getstatic_a 2;        // com/riscure/demo/bach/pin/sensitive/CLASS0.STATICFIELD0
                        invokeinterface 1 3 8;        // resetAndUnblockState org/globalplatform/CVM
                        pop;
                        return;
```

```
74:39 Configuration: CapVerificationTest  Device: Java PC/SC: Broadcom Corp Contacted SmartCard 0
```

Figure 13: Reverse engineering a regenerated JCA file