whiteCryption

Java Code Protection 2.18

User Guide

This software and any associated documentation is provided to you pursuant to the agreement you entered into with whiteCryption Corporation.

Copyright Information

Contact Information

whiteCryption Corporation, 920 Stewart Drive, Suite 100, Sunnyvale, California 94085, USA

contact@whitecryption.com

www.whitecryption.com

Table of Contents

1 Introduction

This chapter provides basic information about Java Code Protection.

1.1 What Is Java Code Protection?

Java Code Protection (JCP) is a command-line tool that converts Java source code of an Android application into native code, and applies a number of software protection features for better security, such as obfuscation, integrity protection, anti-debug, and binary packing.

The end result is a transformed Android application that is functionally equivalent to the original one, but at the same time it is strongly protected against reverse engineering, debugging, tampering, and malware.

The technique used by JCP is based on Cryptanium Code Protection — a comprehensive code protection solution intended for hardening the native C/C++/Objective-C code of software applications on multiple target platforms. This document assumes that you are familiar with basic concepts of Cryptanium Code Protection. You can find detailed information in the *Cryptanium Code Protection User Guide* or by visiting www.whitecryption.com/code-protection.

1.2 Prerequisites

Before protecting your target application with JCP, you must set up the following software on your workstation:

- Java 1.7 or later
- Android SDK for building managed (Java) code
- Android NDK (version r9c or later) for building native (C++) code

  JCP requires Android NDK even if your application's original code does not contain native (C++) code.
- Apache Ant or Gradle for integrating JCP

  JCP requires Ant or Gradle even if you are using a different development environment.

1.3 Limitations

Please carefully review the following list of current JCP limitations before building protected applications:

- If you use Gradle to build your application with JCP, you have to use a Gradle command that builds only one build variant at a time. A build variant is a combination of a build type and a build flavor. If you build several variants at once, certain JCP files and data may get overwritten or reused, which in turn may cause unexpected behavior.

  Assume your build configuration has two build types — "release" and "debug", and there are no build flavors defined. Then if you use the "`gradle build`" command, Gradle will build two variants of your application — one for each build type. In this case, when building the profiling or protected application, JCP files and data from both variants will be mixed together causing unexpected behavior. A correct approach would be to use the "`gradle assembleDebug`" or "`gradle assembleRelease`"

command to build a specific variant of your application. Every build variant has to be profiled and protected separately.

- JCP does not support Android projects that require the ability to generate more than one DEX file (multidex functionality). This means that JCP cannot protect applications that have more than 65536 methods defined. This is a known problem that may happen when using the Google Play Services libraries. A possible solution to this problem is to make sure your application has dependencies on only those packages that you actually need.
- JCP cannot compile Android library (`.aar`) projects.
- JCP does not support the experimental Gradle plugin.
- If you are using Gradle plugin 1.5.0, and do not have ProGuard enabled, and have custom transforms defined using the Transform API, the transforms will be operating on the protected code generated by JCP. Potentially, this may break your transforms.

2 Setting Up JCP

This chapter describes procedures that you have to complete to set up JCP on your workstation and prepare the target application.

2.1 Extracting the JCP Package

JCP is delivered to you as an archived package. You should simply extract it into any folder on your computer. The package contains all the necessary components; you do not have to install Cryptanium Code Protection to use JCP.

2.2 Specifying the Path to Android NDK

JCP needs to know where Android NDK is located, even if your application does not contain native code. Here is how you can do this:

- Create an environment variable named `ANDROID_NDK`, which contains the path to the Android NDK root folder.
- If you are using Ant, append the following command-line argument to the "`ant debug`" or "`ant release`" command when building the target application:

  `-Djcp.ndk.root="«path to the Android NDK root folder»"`
- If you are using Gradle, append the following command-line argument to the Gradle build command when building the target application:

  `-Pjcp.ndk.root="«path to the Android NDK root folder»"`

  You can also specify the path as a property at the bottom of your `build.gradle` file as follows:

  `project.ext.set("jcp.ndk.root", "«path to the Android NDK root folder»")`

2.3 Preparing the Target Application With Ant

To protect your application with JCP and Ant, a few simple modifications need to be made to your project files. To do this, execute the following steps:

1. If the `build.xml` file is not present in target application's root folder, or if it should be updated, execute the following command in the application's root folder:

   ```
   android update project -p . -t «API level»
   ```

   where *«API level»* is the Android API level you use, such as "`android-16`" or "`android-22`".

   This step will create the `build.xml` file (if not present) or update it. For more information on this operation, see developer.android.com/tools/projects/projects-cmdline.html.
2. Open the `build.xml` file in any text editor and add the following line inside the `<project>` tag:

```
<import file="«path to JCP»/jcp_rules.xml" />
```

where *«path to JCP»* is the folder where you extracted the JCP package (see §2.1).

The added line should be placed after the following line (if present):

```
<import file="custom_rules.xml" optional="true" />
```

but immediately before the following line:

```
<import file="${sdk.dir}/tools/ant/build.xml" />
```

The following is an example snippet of a correctly configured `build.xml` file:

```
<import file="custom_rules.xml" optional="true" />
<import file="C:/jcp/jcp_rules.xml" />
<import file="${sdk.dir}/tools/ant/build.xml" />
```

3. Optionally, enable ProGuard for your application.

   Although not mandatory, this will significantly improve the obfuscation level by renaming internal class, method, and field names. For information on ProGuard, see developer.android.com/tools/help/proguard.html.

2.4 Preparing the Target Application With Gradle

To protect your application with JCP and Gradle, a few simple modifications need to be made to your project files. To do this, execute the following steps:

1. Open the `build.gradle` file (by default, located in the `app` folder) in any text editor and add the following line at the bottom of the file:

   ```
   apply from: '«path to JCP»/jcp.gradle'
   ```

   where *«path to JCP»* is the folder where you extracted the JCP package (see §2.1).

   The following is an example of a correctly configured `build.gradle` file:

   ```
   apply plugin: 'com.android.application'

   android {
       compileSdkVersion 22
       buildToolsVersion "22.0.1"
       defaultConfig {
           applicationId "com.example.myapplication1"
           minSdkVersion 15
           targetSdkVersion 22
           versionCode 1
           versionName "1.0"
       }
       buildTypes {
           release {
   ```

```
            minifyEnabled true
            proguardFiles getDefaultProguardFile('proguard-android.txt'),
                'proguard-rules.pro'
        }
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:appcompat-v7:22.1.1'
}

apply from: 'C:\\JCP\\jcp.gradle'
```

2. Optionally, enable ProGuard for your application.

   Although not mandatory, this will significantly improve the obfuscation level by renaming internal class, method, and field names. For information on ProGuard, see the following page: developer.android.com/tools/help/proguard.html

3 Profiling the Target Application

This chapter describes how to optimize execution speed of the target application by executing the profiling process.

3.1 What Is Profiling?

Profiling is a process that allows JCP to automatically analyze target application's run-time behavior to detect its speed-sensitive functions. Functions that are most frequently called are usually not critical to application's security and therefore can have a reduced protection level applied, resulting in faster execution speed.

> ⚠ While profiling is not mandatory, we strongly recommend profiling the applications you protect with JCP. Otherwise, the application's execution speed may be slow.

Profiling takes place as follows (shown in Figure 3.1):

1. JCP builds a modified version of the target application, which contains specific profiling code embedded into the source code.

   The built application is called the profiling application.

2. A developer installs and runs the profiling application on a target device.

   While running, the profiling application saves profiling data to a local file on the device.

3. When the profiling application ends it execution, JCP downloads and imports the profiling data file.

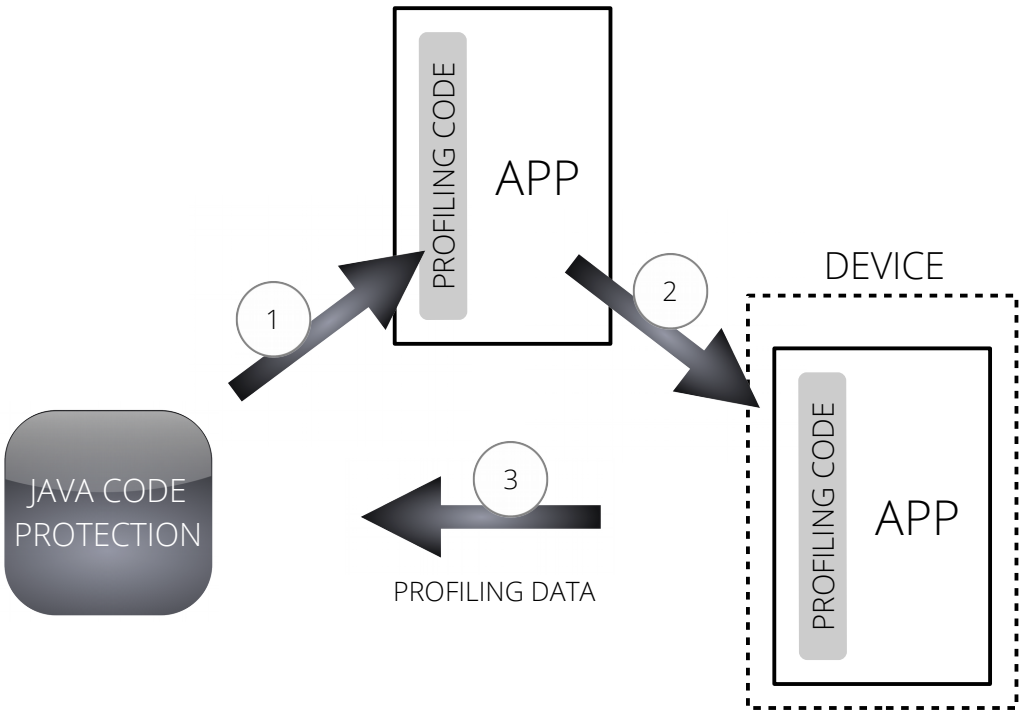

Figure 3.1: Profiling an application

JCP uses the profiling data to automatically adjust the security level of the target application when building the protected version.

> ⚠ Normally, you do not have to repeat profiling if you make insignificant changes to the source code (if there are no new functions created). This, however, is not the case if your application uses ProGuard. Then, even very small changes, like adding a new class to the project, will make profiling statistics useless because all the names of functions will be different. In such a case, you must run the profiling process again.

3.2 Executing the Profiling Process With Ant

To build the profiling application with Ant and execute profiling, proceed as follows:

1. Make sure you have completed all procedures described in §2.
2. Add the following permissions to the `AndroidManifest.xml` file directly under main `<manifest>` tag (if not present):

   ```
   <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
   <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
   ```

   These permissions are required only during the profiling process when the profiling application saves the statistics file to the external storage. You can safely remove these permissions when building the final protected application, unless you need them for other purposes.
3. Open the command-line prompt and navigate to the root folder of your application's source code.

   This folder should contain subfolders `src`, `res`, and `jni` (if your application contains native code) and the `AndroidManifest.xml` file.
4. If your application contains native code, build it by executing the "`ndk-build`" command.
5. To build the profiling application, execute either the "`ant debug`" or "`ant release`" command but make sure you append the "`-Djcp.build.profiling=true`" argument to it.

   For example, if you are building the debug edition of the profiling application, execute the following command:

   ```
   ant debug -Djcp.build.profiling=true
   ```

   > ⚠ If you are about to build a protected release version of your application and the application uses ProGuard, then you have to also build a release version of the profiling application using the "`ant release`" command because, with ProGuard, the function names in the release version will be completely different from the debug version.

   Optionally, you can append other properties to the build command to customize your application as described in §5.

   Once built, the profiling application will be packed into an `.apk` file.
6. If you have built the release version of the application, sign it using your developer's key as described in developer.android.com/tools/publishing/app-signing.html.

7. Install the `.apk` file on an Android device.
8. If the device is running Android 6.0 (API 23) or later, and your application is targeting API 23 or later, explicitly enable the permission for the profiling application to write to the external storage using one of the following methods:
   - Open global settings, select your profiling application, and, under **Permissions**, enable the permission to write to the external storage.
   - Grant the permission to read and write to the external storage from the command line using the `pm grant` command.
     For example, if you are using ADB, this can be achieved as follows:

     ```
     adb shell pm grant «package name» android.permission.READ_EXTERNAL_STORAGE
     adb shell pm grant «package name» android.permission.WRITE_EXTERNAL_STORAGE
     ```

     where *«package name»* is the name of your profiling application's package.
9. Run the profiling application on the device and work with the profiling application on the device in a manner similar to real-life usage.

   Such behavior will provide the best profiling statistics to JCP and will result in an optimal balance between speed and security.
10. Stop the profiling application on the device.
11. To load the profiling statistics, execute the following command:

    ```
    ant load-profiling
    ```

    This script downloads the profiling data file from the device and instructs JCP to import the file.

3.3 Executing the Profiling Process With Gradle

To build the profiling application with Gradle and execute profiling, proceed as follows:

1. Make sure you have completed all procedures described in §2.
2. Add the following permissions to the `AndroidManifest.xml` file directly under main `<manifest>` tag (if not present):

   ```
   <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
   <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
   ```

   These permissions are required only during the profiling process when the profiling application saves the statistics file to the external storage. You can safely remove these permissions when building the final protected application, unless you need them for other purposes.
3. Open the command-line prompt and navigate to the root folder of your application's source code.

   This folder should contain the `app` folder and the `build.gradle` file.
4. To build the profiling application, execute the build command, but make sure you append the "`-Pjcp.build.profiling=true`" argument to it, for example as follows:

```
gradle assembleDebug -Pjcp.build.profiling=true
```

> ⚠ Do not use a Gradle command that builds several build variants of your application at once, because this may result in overwriting or reusing of some JCP files and data. You have to build and profile every build variant separately. For more information on this restriction, see §1.3.

> ⚠ If you are about to build a protected release version of your application and the application uses ProGuard, then you have to install and profile the release version of the application because, with ProGuard, the function names in the release version will be completely different from the debug version.

Optionally, you can append other properties to the build command to customize your application as described in §5.

Once built, the profiling application will be packed into an `.apk` file.

5. If you have built the release version of the application, sign it using your developer's key as described in developer.android.com/tools/publishing/app-signing.html.
6. Install the `.apk` file on an Android device.
7. If the device is running Android 6.0 (API 23) or later, and your application is targeting API 23 or later, explicitly enable the permission for the profiling application to write to the external storage using one of the following methods:
   - Open global settings, select your profiling application, and, under **Permissions**, enable the permission to write to the external storage.
   - Grant the permission to read and write to the external storage from the command line using the `pm grant` command.
     For example, if you are using ADB, this can be achieved as follows:

     ```
     adb shell pm grant «package name» android.permission.READ_EXTERNAL_STORAGE
     adb shell pm grant «package name» android.permission.WRITE_EXTERNAL_STORAGE
     ```

     where *«package name»* is the name of your profiling application's package.
8. Run the profiling application on the device and work with the profiling application on the device in a manner similar to real-life usage.

   Such behavior will provide the best profiling statistics to JCP and will result in an optimal balance between speed and security.
9. Stop the profiling application on the device.
10. To load the profiling statistics, execute the following command:

    ```
    gradle loadProfiling
    ```

    This script downloads the profiling data file from the device and instructs JCP to import the file.

4 Protecting the Application

This chapter describes how to build and customize the final protected version of the target application.

4.1 Building the Protected Application

To build the final protected application, proceed as follows:

1. Make sure you have completed all procedures described in §2.
2. Execute the profiling process as described in §3.

   Profiling is not mandatory, but strongly recommended.
3. Open the command-line prompt and navigate to the root folder of your application's source code.

   If you're building with Ant, this folder should contain folders `src`, `res`, and `jni` (if your application contains native code) and the `AndroidManifest.xml` file.

   If you're building with Gradle, this folder should contain the `app` folder and the `build.gradle` file.
4. Remove the `READ_EXTERNAL_STORAGE` and `WRITE_EXTERNAL_STORAGE` permissions from the `AndroidManifest.xml` file if you added them for profiling purposes.
5. If you are using Ant and your application contains native code that you have not already built, build the native code by executing the "`ndk-build`" command.
6. Build the target application as follows:
   - If you are using Ant, build the target application using either the "`ant debug`" or "`ant release`" command, depending on the build type.
   - If you are using Gradle, use a build command that builds only one build variant of your application.

     > ⚠ Using a Gradle command that builds several build variants of your application at once may cause overwriting or reusing of some JCP files and data. You have to build and protect every build variant separately. For more information on this restriction, see §1.3.

   Optionally, you can append certain properties to the build command to customize your application as described in §5.

   > ⚠ If you are building a protected release version of your application and the application uses ProGuard, then you must make sure the profiling data came from a release version of the profiling application because, with ProGuard, the function names in the release version will be completely different from the debug version.

   Once built, the protected application will be packed into an `.apk` file, which you can distribute to your customers.
7. If you have built the release version of the application, sign it using your developer's key as described in developer.android.com/tools/publishing/app-signing.html.

4.2 Excluding Classes From Processing

JCP provides a mechanism for excluding specific classes or entire packages from processing. For example, this may be useful if some classes are very large and slow down the protection process, or if (in very rare cases) some particular classes cause the build system to crash.

If a class is excluded from processing, JCP will not modify it at all and will not include any protection features in them.

> ⚠ You should avoid excluding classes if possible, because unprocessed classes will essentially be unprotected.

To exclude specific classes from processing, proceed as follows:

1. Detect the large files that are slowing down the process using the following methods:
   - In the build output, notice lines similar to the following that take a long time to complete:

     ```
     [exec] "Compile arm  : scp <= jcp1475.c
     ```

     For instance, if the preceding line takes a long time to complete, this means that `jcp1475.c` is a really large file and should be excluded from obfuscation.
   - Alternatively, go to the `jni-jcp_protected` folder in the application's source folder, sort the `.c` files by size, and pick the largest files.
2. Once you have selected the large files to be excluded, find the corresponding Java classes for each file.

   You can do this by opening the unprotected version of the `.c` file (in the `jni-jcp` folder) in a text editor. The Java class name will be displayed at the very beginning of the file.

   > ⚠ The Java class name will not be visible in `.c` files that are located in the `jni-jcp_profiling` or `jni-jcp_protected` folders.
3. Exclude the Java classes from processing using one of the following methods:
   - Use a text file named `jcp-exclude.txt` that lists all Java classes and packages to be excluded.

     If you are using Ant, you must manually create the `jcp-exclude.txt` file and place it in your application's root folder. If you are using Gradle, you must use the `jcp-exclude.txt` file that is already provided in the JCP package (see §2.1).

     In the `jcp-exclude.txt` file, you must use the same format as used in the `.c` file, and place every class on a new line, for example as follows:

     ```
     android/support/v4/widget/ViewDragHelper
     android/support/v4/widget/DrawerLayout
     android/support/v4/widget/SlidingPaneLayout
     ```

     You can also specify entire Java packages to be excluded in a similar manner. For instance, the following line will instruct JCP to exclude all classes in the `android.support.v4.widget` package:

     ```
     android/support/v4/widget
     ```

     By default, JCP will exclude all nested classes as well. If you want to keep the nested classes protected, you must set the "`jcp.exclude.nested argument`" to 0 as described in §5.

- Alternatively, if you have included the `jcp-cryptanium.jar` library in your `libs` folder, you can add the following annotation directly into the Java source code of the large class files:

  ```
  @com.cryptanium.annotation.Obfuscate(false)
  ```

  This annotation should be placed immediately before the class definition, for example as follows:

  ```
  @com.cryptanium.annotation.Obfuscate(false)
  public class MyLargeClass extends ...
  ```

  The `jcp-cryptanium.jar` file is available in the JCP package.
  If you are using ProGuard, you must tell ProGuard not to obfuscate these annotations. Simply add the following lines to the end of the `proguard-project.txt` file, which is located in your application's root folder:

  ```
  -keep @interface com.cryptanium.annotation.*
  -keepclassmembers @interface com.cryptanium.annotation.* {
      public *;
  }
  ```

4. Build the protected application again as described in §4.1.

5 Configuration Properties

This chapter describes configuration properties that you can append as parameters to build commands to customize the target application.

5.1 Using the Configuration Properties

If you are using Ant, all appended parameters must be preceded with "`-D`", for example "`ant release -Djcp.abi="x86"`".

If you are using Gradle, all appended parameters must be preceded with "`-P`", for example "`gradle assembleRelease -Pjcp.abi="x86"`".

5.2 Supported Properties

JCP supports the following configuration properties:

`jcp.ndk.root`

This property specifies the path to the NDK installation folder (see §2.2).

`jcp.ndk.cpu`

This property specifies how many CPU cores should be used for building the target application.

With Ant, the default value is 2. With Gradle, the default value is the number of available cores.

`jcp.skip.scp`

This property allows you to enable or disable all protection features (such as obfuscation, integrity protection, anti-debug, and binary packing) for the target application. Regardless of the value specified, JCP will still convert Java code to native code.

Possible values are `false` (default), which enables the protection features, and `true`, which disables the protection features.

`jcp.enable`

This property allows you to completely disable JCP protection (including converting Java code to native code) for the target application. Consequently, the target application is built as if JCP does not exist.

Possible values are `true` (default), which enables JCP protection, and `false`, which disables the protection.

`jcp.build.profiling`

If this property is set to `true`, JCP will embed profiling functions into the target application (see §3).

The default value is `false`.

`jcp.strength`

This property specifies protection level applied to the target application. The following values can be specified:

- `-jcp.strength=1`: maximum speed and least security
- `-jcp.strength=2`: default value, which provides a good balance between speed and security
- `-jcp.strength=3`: maximum security and least speed

`jcp.abi`

This property specifies the architectures to be used for the converted native code.

This is not important when the Android virtual machine runs Java bytecode only. With JCP, when the Java code is converted to native code, you have to decide which architectures will be supported by your application. The following values are available:

- `armeabi` (default)
- `armeabi-v7a`
- `arm64-v8a`
- `x86`
- `x86_64`
- `mips`

You can specify multiple architectures by separating them with space and putting the entire list between double quotes, for example "`jcp.abi="armeabi x86"`".

`jcp.exclude.nested`

When certain classes are excluded from processing (see §4.2), this property specifies whether nested classes should also be excluded.

The default value is 1, which means that nested classes must be excluded. To avoid excluding nested classes, set this parameter to 0.

`jcp.short.commands`

This property corresponds to the `APP_SHORT_COMMANDS` flag of the `Application.mk` file. You should set it to `true` if the application fails to build due to a very long list of sources and dependent libraries in your project.

The default value is `false`.

`jcp.ndk.verbose`

This property corresponds to the `V` flag of the NDK build system. Setting this property to 1 will produce more information in the NDK build output.

The default value is 0.

`jcp.protect.libs`

This property specifies whether JCP should also protect libraries in the `lib` folder and library projects added as references to your current project.

Because ProGuard recursively includes referenced libraries in the code, you must be aware that if you set this property to `false` and use ProGuard in the release mode, some classes that are protected in the release mode may not get protected in the debug mode.

The default value is `true`.

`jcp.output.soname`

By default, the protected native library, which contains code converted from Java to C, is named `scp.so`. This may cause naming conflicts if there are several protected libraries in use. By using this property, you can specify a different name for the protected native library.

`jcp.proguard.mapping`

If enabled, ProGuard renames classes for obfuscation purposes. If you have configured JCP to exlude certain classes from processing (see §4.2), JCP uses the ProGuard mapping file to determine which original class corresponds to which renamed class. By default, JCP tries to find the ProGuard mapping file automatically. However, if the mapping file is located in a non-standard place, JCP may fail to find it. If this is the case, you must provide the `jcp.proguard.mapping` property whose value must contain the path to the ProGuard mapping file.