Artifacts for ACSAC'24 paper "Manifest Problems: Analyzing Code Transparency for Android Application Bundles"

Abstract

The artifacts submitted for our paper include custom software and data we used for carrying out the large-scale analyses of CT for AAB described in the Evaluation section (Section 7.1), as well as the manipulated app binary for the case study in Section 7.2. All software is either supplied as Java source code or Bash scripts. All Java projects include gradle build scripts. Please note that executing the large-scale evaluation of apps on Google Play takes multiple days due to rate-limiting of the AndroZoo server. The evaluation of apps on Huawei AppGallery takes multiple hours on a mid-range laptop. For both processes, we include intermediary and final results as data artifacts.

Tested Environments

We used the following environment when working on this project:

- **JDK**: openjdk 22.0.2
- **IDE**: IntelliJ Community Edition 2024.2.1
- **Android Studio**: 2024.1.2
- **OS**: macOS Sonoma 14.5 (23F79) (x86_x64)

The following environment has also been confirmed to work (kudos to our anonymous artifact reviewers at ACSAC'24!):

- **JDK**: Temurin 17.0.12
- **IDE**: IntelliJ Community Edition 2024.2.1
- **Android Studio**: 2024.1.2.12
- **OS**: Ubuntu 24.04 LTS (x86_x64)

Please Note: Some parts of this project require a decent amount of RAM. If execution fails with an OutOfMemoryException, increase the maximum heap allowed to the JVM, e.g. by passing -Xmx8192M as an additional argument to the java command.

Artifact Details

The artifact is self-contained. Every folder in the repository contains a README.md file explaining the purpose and usage of the respective artifact. We additionally provide a compilation of all documentation below.

0.1. Code

Custom tooling and code we developed for our evaluation can be found in the `Code` folder of the artifact repository.

0.0.1. Case Study: Microsoft Loop

This folder contains all the code (and binaries) needed for reproducing the Microsoft Loop case study in paper Section 7.2.1. As documented in the paper, the attack injects a dependency on a static library into the manifest of the Microsoft Loop app. As a result, the classes of the static library are added to the app's class path at app launch, which effectively grants the attacker code execution in the context of the victim app. The static library contains a class named identical to the Microsoft Loop app's main activity. When the manipulated Loop app is launched, the class from the static library takes precedence over the app's original main activity. Our fake main activity displays a simply login screen that illustrates the potential to use this attack to extract login credentials.

0.0.1.1. Prerequisites

- Install bundletool and make sure it is in your PATH, e.g. by following the instructions from https://stackoverflow.com/a/67332078

0.0.1.2. The experiment

1. Check the Code Transparency of the original APKs in the Original_APKs folder:

   `../../Scripts/check_ct.sh Original_APKs/base.apk`

Output:

```
...
Code transparency signature is valid. SHA-256 fingerprint of the code transparenc
...
```

2. Check the Code Transparency of the patched APKs in the Patched_APKs folder:

   `../../Scripts/check_ct.sh Patched_APKs/com.microsoft.loop/base.apk-signed.apk`

Output:

```
...
Code transparency signature is valid. SHA-256 fingerprint of the code transparenc
...
```

Please Note: If you get an exception that says: "Invalid CEN header (invalid zip64 extra data field size)", please make sure you use a JRE that includes a fix for https://bugs.openjdk.org/browse/JDK-8313765 !

3. Install the static library on a development-enabled Android device connected via USB (we used a Google Pixel 3 running Android 12):

   ```
   adb install static_library.apk
   ```

4. Install the patched APKs for the Microsoft Loop app:

   ```
   adb install-multiple Patched_APKs/com.microsoft.loop/*.apk
   ```

5. Launch the Microsoft Loop app on the Android device

6. You are greeted with the fake Login screen from the static library (see Fake_Login_Screen.png). Despite the attacker having code execution in the context of the Microsoft Loop app, the CT still validates as if the app was untampered.

Please Note: We do not have access to the private key for the APK's app signing certificate, so we cannot entirely accurately simulate a supply chain attacker. In a real-world attack, the manipulated APK would be signed with the developer's correct app signing certificate. As explained in paper Section 3, the AAB scheme requires the developer to share the app signing key with the distributor.

0.0.1.3. Building the individual Case Study components

0.0.1.3.1. Prerequisites

- Download and extract the A2P2 Patching Pipeline distribution from https://extgit.iaik.tugraz.at/fdraschbacher/a2p2.The most recent distribution bundle as of this writing is https://extgit.iaik.tugraz.at/fdraschbacher/a2p2/-/blob/main/a2p2_distribution_v1.0.1.zip?ref_type=heads.
- IntelliJ IDEA (Community Edition)

0.0.1.3.2. Building the A2P2 stage

1. Open the A2P2_Stage_Sources project in IntelliJ IDEA (Community Edition)
2. Ensure the A2P2 path in build.gradle points to the extracted A2P2 distribution on your system
3. In the gradle pane, double-click on JAR to build the JAR file
4. The resulting JAR is located at `build/libs/A2P2_Stage_Sources.jar`

0.0.1.3.3. Building the Static Library

1. Open the Static_Library_Sources folder in Android Studio
2. Build like you would any other Android project

0.0.1.3.4. Patching the Microsoft Loop app

0. If you haven't already, download and extract the A2P2 distribution from https://extgit.iaik.tugraz.at/fdraschbacher/a2p2/-/blob/main/a2p2_distribution_v1.0.1.zip?ref_type=heads
1. Export the A2P2 distribution folder's path: `export A2P2_PATH=your/path/here`
2. In the extracted A2P2 distribution folder, create a folder named "stages"
3. Copy the A2P2 stage JAR file ("a2p2_stage_inject_static_library.jar") to that folder
4. Run A2P2:

```
mkdir output
java -jar $A2P2_PATH/a2p2.jar {./Original_APKs} ! unpack !
injectstaticlibrary com.loop.patch.library 1
3D2225686339F019C49C8111333ECBF7B877A158E17BD439B3E899AB42F6DBCF ! pack !
sign ! ./output
```

5. The patched APKs will be located in `output/com.microsoft.loop/`

0.0.2. App Analyzer

The Java source code for the tool we used in Sections 7.1.4 and 7.1.5 for evaluating the susceptibility of an app to attacks A4 and A5. It scans the APK file for files in the assets or resources that carry the file signature or extension of the ELF, DEX, APK, or DLL formats (A4). Additionally, all files in these folders that contain only printable characters are ran through the Esprima JavaScript syntax validator. Lastly, our tool checks the app's DEX files for the presence of the FileProvider class of the Jetpack Core library (A5).

0.0.2.1. Prerequisites:

- Working JDK installation and JRE
- IntelliJ IDEA (e.g. the free community edition)

0.0.2.2. Steps to run

1. Open the gradle project in IntelliJ IDEA
2. Open the Main.java file
3. Right-click the green triangle next to "public class Main" in line 200
4. Click on "Modify Run Configuration..."
5. Enter the input and output paths as the first and second argument in the Program arguments text field The input path must point to a valid AppSet folder structure (see AppSet_MostPopularGooglePlay or AppSet_UsingCodeTransparency for examples). Inside

the root folder of the AppSet, a set of category folders contains one folder per application, which contains the APKs for that app.

6. Click on Apply -> OK
7. Launch the run configuration through the green triangle in the top toolbar

The result of this process is a CSV file that contains these fields for each app:

- packageName: The package name of the app
- category: The app category as stored in the app set
- aab: Whether or not the app was compiled from an AAB
- usesFileProvider: The app's manifest exposes a FileProvider
- hasFileProvider: The app contains the FileProvider class in its DEX code
- nsc: The app contains a network configuration xml file referenced in its manifest
- ct: The APK contains a Code Transparency JWT file
- dexFiles: A list of dex files found in non-standard locations in the APK
- elfFiles: A list of elf files found in non-standard locations in the APK
- apkFiles: A list of APK files found in the APK
- jsFiles: A list of Javascript files found in the APK
- dllFiles: A list of DLL files found in the APK (used by apps written in C# or similar)
- htmlFiles: A list of html files found in the APK

0.0.3. Google Play Crawler and CT Checker

The Java source code for the tool we used in Section 7.1.1 for scanning Google Play for apps that use the AAB distribution format and apps that use Code Transparency. Since no official list of apps on Google Play or API for downloading APKs exists, we use the AndroZoo dataset as an index and for obtaining APKs. All apps are checked for whether they were compiled from an Android Application Bundle, and whether they contain a Code Transparency JWT file. For speeding up this task, only the ZIP Central Directory of each APK is actually retrieved from the AndroZoo repository server.

0.0.3.1. Prerequisites:

- Working installation of JDK 17 or later
- IntelliJ IDEA (e.g. the free community edition)
- Downloaded and unpacked AndroZoo index file from https://androzoo.uni.lu/static/lists/latest.csv.gz

0.0.3.2. Steps to run

0.0.3.2.1. A. Assemble list of apps

1. Open this gradle project in IntelliJ IDEA

2. Open the AndroZooPackageNameCrawler.java file
3. Right-click the green triangle next to "public class AndroZooPackageNameCrawler" in line 10
4. Click on "Modify Run Configuration..."
5. Enter the full path of the unpacked (csv) AndroZoo index file as the first argument in the Program arguments text field
6. Enter your desired output path (where you wish results to be placed) as the second argument in the Program arguments text field
7. Click on Apply -> OK
8. Launch the run configuration through the green triangle in the top toolbar

The result of this process is a JSON file that contains the package names for all Google Play apps in the AndroZoo dataset.

0.0.3.2.2. B. Retrieve app metadata from Google Play

1. Open this gradle project in IntelliJ IDEA
2. Open the AppDetailDownloader.java file
3. Right-click the green triangle next to "public class AppDetailDownloader" in line 12
4. Click on "Modify Run Configuration..."
5. Enter the output path of the AndroZooPackageNameCrawler as the first argument in the Program arguments text field
6. Enter your desired output path (where you wish results to be placed) as the second argument in the Program arguments text field
7. Click on Apply -> OK
8. Launch the run configuration through the green triangle in the top toolbar
9. Expect this to run a few hours

The result of this process is a set of JSON files that contain metadata for all Google Play apps in the AndroZoo dataset.

0.0.3.2.3. C. Check Code Transparency of apps

1. Open the CodeTransparencyChecker.java file
2. Set the ANDROZOO_INDEX_PATH static field to point to your AndroZoo index file (latest.csv)
3. Right-click the green triangle next to "public class CodeTransparencyChecker" in line 11
4. Click on "Modify Run Configuration..."
5. Set the input and output paths in the Program arguments text field Make sure the input path matches the output path of the AppDetailDownloader run
6. Click on Apply -> OK
7. Launch the run configuration through the green triangle in the top toolbar
8. Expect this to run a few days

The result of this process is a set of JSON files that contain AAB and CT status for all apps on Google Play. Every file contains a JSON object whose keys are package names for applications. The value for each key is the AAB and CT status. 1 indicates that the app was compiled from an AAB, while 3 indicates that it additionally uses CT.

0.0.4. Huawei App Gallery Crawler and CT Checker

Tthe Java source code for the tool we used in Section 7.1.1 for scanning Huawei AppGallery for apps that use the AAB distribution format and apps that use Code Transparency. Since no official list of apps on Huawei AppGallery exists, it first brute-forces app identifiers through Huawei's Rest API to collect a list of known package names. Next, all these apps are checked for whether they were compiled from an Android Application Bundle, and whether they contain a Code Transparency JWT file. The tool only obtains the ZIP Central Directory of each APK from Huawei's servers to speed up the analysis.

0.0.4.1. Prerequisites:

- Working JDK installation
- IntelliJ IDEA (e.g. the free community edition)

0.0.4.2. Steps to run

0.0.4.2.1. A. Brute-force application identifiers

1. Open this gradle project in IntelliJ IDEA
2. Open the PackageNameCrawler.java file 2b. Adapt id ranges (see below)
3. Right-click the green triangle next to "public class PackageNameCrawler" in line 12
4. Click on "Modify Run Configuration..."
5. Enter the output path (where you wish results to be placed) in the Program arguments text field
6. Click on Apply -> OK
7. Launch the run configuration through the green triangle in the top toolbar

The result of this process is a set of JSON files that contain metadata of all apps on Huawei AppGallery. Please note we ran the brute-forcing multiple times, covering different identifier ranges (see below).

0.0.4.2.2. B. Check Code Transparency of apps

1. Open the CodeTransparencyChecker.java file
2. Right-click the green triangle next to "public class CodeTransparencyChecker" in line 11
3. Click on "Modify Run Configuration..."
4. Set the input and output paths in the Program arguments text field The last argument is the output paths The first n-1 arguments are the input paths Make sure each input path

matches the output path of a PackageNameCrawler runs Multiple PackageNameCrawler runs are needed to cover multiple valid app identifier ranges (see below).
5. Click on Apply -> OK
6. Launch the run configuration through the green triangle in the top toolbar

The result of this process is a set of JSON files that contain AAB and CT status for all apps on Huawei AppGallery. Every file contains a JSON object whose keys are package names for applications. The value for each key is the AAB and CT status. 1 indicates that the app was compiled from an AAB, while 3 indicates that it additionally uses CT.

0.0.4.2.3. A note on Huawei AppGallery app identifiers

App identifiers start with letter C, followed by a number that is either 4-6 or 9 digits long. Not all numbers following this pattern are valid app identifiers. It seems there are multiple different ranges of valid identifiers within this total space of identifiers. For our evaluation, we identified these ranges through trial-and-error, and ran the PackageNameCrawler once for each range, by adapting the idRange and idOffset static fields. We then fed all the results folders into CodeTransparencyChecker.

The different ranges we identified are (broadly): Low: C2955-C100940 Most: C100308949-C109981511 High: C110027587-C111000491

0.0.5. Results Summarizer

This folder contains the Java source code for the tool we used in Sections 7.1.4 and 7.1.5 for summarizing the results of the CT and AAB analyses carried out in GooglePlayCrawler and HuaweiAppGalleryCrawler.

0.0.5.1. Prerequisites:

- Working JDK installation
- IntelliJ IDEA (e.g. the free community edition)

0.0.5.2. Steps to run

1. Open this gradle project in IntelliJ IDEA
2. Open the ResultsSummarizer.java file
3. Right-click the green triangle next to "public class ResultsSummarizer" in line 3
4. Click on "Modify Run Configuration..."
5. Enter the input path in the Program arguments text field The input path must point to the output folder of the CodeTransparencyChecker in GooglePlayCrawler or HuaweiAppGalleryCrawler.
6. Click on Apply -> OK
7. Launch the run configuration through the green triangle in the top toolbar

The program prints a summary of the results to stdout. The summary includes a total of apps that use AAB and of apps that use CT.

0.0.6. Utility scripts for working with AAB and CT

These utilities speed up common tasks when working with bundletool, Google's official Code Transparency implementation.

0.0.6.1. add_ct.sh

Adds a Code Transparency to an AAB file. Usage: add_ct.sh {aab_path} {keystore_path}

0.0.6.2. build_apks.sh

Builds signed APKs from an AAB file. Usage: build_apks.sh {aab_path} {keystore_path} {keystore_pass}

0.0.6.3. check_ct.sh

Checks the Code Transparency of an APK file. Usage: check_ct.sh {apk_path}

0.0.6.4. gen_ct_cert.sh

Generates a keystore for use in generating a CT. Usage: gen_ct_cert.sh {password} {file_path}

0.1. Data

Contains intermediary and final results of the evaluation of our paper. Many of the files were generated using the tools contained in the Code folder of the artifact repository.

1. AppSet_MostPopularGooglePlay

The dataset of most popular applications from Google Play (as of December 2023) we used in Section 7.1.5.

2. AppSet_UsingCodeTransparency

All apps we identified as using Code Transparency during our analyses. Referenced in Sections 7.1.2 and 7.1.3.

3. Evaluation_GooglePlay

Intermediary and final results of the large-scale AAB and CT analysis of (close to) all apps from Google Play. See Section 7.1 of the paper.

4. Evaluation_HuaweiAppGallery

Intermediary and final results of the large-scale AAB and CT analysis of (close to) all apps from Huawei AppGallery. See Section 7.1 of the paper.

5. Evaluation_TopAppsGooglePlay

Detailed analysis results for the susceptibility of top apps from Google Play regarding CT attacks. See Section 7.1.5 in the paper.