**Week 8: Analyze an Open Source Project**

Nate Bachmeier

April 28th, 2019

TIM-8101: Principals of Computer Science

Northcentral University

# Bottoms up Analysis using F-Droid

F-Droid is an open source mobile application store for the Android platform. It allows developers to share source code and an easy model for side loading compiled versions of the applications onto physical devices.

Normally open source repositories are analyzed from the top-down, however this assumes one even knows what is relevant to the final binary. Instead a macro bottoms up solution was performed across three passes. The first pass downloaded 1610 APK files and extracted high level features. Next decomplication was performed on 339 files, and finally specific interesting characteristics was explored in one of them.

# Finding Interesting Projects on F-Droid

## Downloading APK Files

F-Droid exposes a complete directory of projects hosted by their platform. Each directory entry has a standardized template which was parsed to get the source code and the compiled Android PacKage (APK). When multiple versions were available, only the most recent was selected for download resulting in 1610 results totaling approximately 6.5gb of binary files.

The median file size was 4.2mb with a standard deviation of 7.8mb and a maximum size of 88.9mb. The ignore the effects of the long tail the 99th percentile was used as the as largest size resulting in 39.8mb.

APK files use the same format as regular ZIP archives and can be extracted with commodity decompression software. The root cause for many of the large APK is too many resource files. Some APKs, like com.dkanada.icecons.apk, include multiple resolutions of the graphics and then select at runtime. Others mitigated this issue by choosing at build time and creating one APK per device category (eg. SmartTV vs mobile phone).

## Opening the APK

Inside of each archive is AndroidManifest.xml, classes.dex, lib folders, and res folders. The manifest describes the permissions and public symbols of the application. Dalvick EXecutable files is compiled code for the virtual machine. The LIBraries folder contains any platform native binaries that need to be consumed through the Java Native Interface. RESources are images, configuration, and layout templates.

Using the dex2jar-2.0 and OpenJDK’s javap utilities a randomly sampled 25% subset of classes.dex files were transformed first into Java Archive format. The archives were expanded to get the Java Class files, which were then converted into Java Assembly files. The Assembly files were then mined to extract a graph representation of the inheritance and call trees. The graphs were then persisted as GraphML files which can be imported into existing graph analysis software.

## Common Themes

Roughly 65% of sampled application are linked against Android Jetpack. Jetpack is removes common boiler plate code and accelerates the time to develop new applications. Developers can use it from scenarios ranging from interacting with SQLite to animating transitions.

ProGuard is an open source tool for obfuscation and code reduction that is used by 23.2% of sampled applications. These projects are easily identified as all private methods are reduced to single character names. A probable reason for obfuscating open source code is to reduce the final binary size and improve initial download latency.

Kotlin is a new language that can be compiled into JavaScript, Java, or into the LLVM compiler framework. Only 8.7% of the sampled applications were written in this modern language. This was surprisingly low given the reduced barrier to entry and marketing push from Google.

Metaprogramming was another theme that was exposed in many of the common packages such as retrofit, butterknife, and dagger, being used by 10% of applications. Traditionally Java has uses reflection however many of the mobile applications leverage Gradle plugins to move the runtime type analysis to compile time.

## Exploring the Manifest

AndroidManifest.xml describes the components of an application and how they are permitted to interact with the system. The primary entities are activities, providers, services, and receivers. An activity defines the UI behaviors; providers share application content; services are long running background code; and receivers are woken up to handle events (called Intents).

Services and receivers create an interesting attack vector as they have privileges and do not require user input. For example, a malicious application could send an exploit inside of an Intent object and execute code in the context of another application. Android’s platform mitigates this scenario by exposing permissions on exported application entities.

To examine this scenario the 1365 of the manifest files were parsed into a graph structure and analyzed with Apache TinkerPop. The graph identified 783 manifests with dangerous permissions; of these 94 service endpoints and 88 receivers are publicly exported and do not require authentication. Performing a deduplication of the declaring manifest results in 119 or 8.7% of sampled applications. If that ratio held true across the Google Play store it would equate to 183,000 vulnerable applications (Statista, 2019).

# Explore a specific application

The application org.smssecure.smssecure.apk has the dangerous permission SEND\_SMS and exposes a vulnerable service endpoint. A manual inspection of the code confirmed that the QuickResponseService is missing the android:permission filter and contains a code path for sending a text message (see figures 2 & 3).

## Describe the High-Level Flow

1. The QuickResponseService is given the intent ACTION RESPOND VIA MESSAGE.
2. The Intent structure contains the phone numbers and text message contents.
3. The service calls MessageSender to store the message in a local outbox database.
4. The MessageSender creates a SmsSendJob to bind the record and send action.
5. The JobManager is receives the job will schedules it for execution.
6. The SmsSendJob will bind to the SmsManger system service.
7. The system API sendMultipartTextMessage is called to send the message.
8. The outbox record and related state are cleaned up.

## Reducing False Positives

One of the challenges with relying solely on the Manifest information to detect vulnerabilities filtering out false positives. If this example code path did eventually call SmsManager then the dangerous permission could not be exploitable.

To reduce these false positives a mechanism is needed to automatically confirm if a misconfigured service will call an API of interest. Since the manifest declares both the service entry class and the applications permissions, it is possible to infer the path of interest. If the path does not exist in the call graph, then it is likely a false positive and can be ignored.

The APK was decompiled into Java Assembly and its call structure mined into a GraphML document. This document was loaded into Apache TinkerPop and the path’s existence confirmed (see appendix).

# Appendix







