ABUSING WEB APIS THROUGH SCRIPTED ANDROID APPLICATIONS

Introduction:

This paper is intended to be a practical walkthrough on scripting Android applications, application package file (APK)s, for arbitrary purposes. We build on the functionality provided by several existing tools and attempt to provide a firm understanding for extending them in the future.

Background:

To begin we must have an understanding of the Android operating system and the way that it executes applications. Android applications are run on a virtual machine known as Dalvik, which is based on the Apache Software Foundations implementation of the Java Virtual Machine, Harmony. These applications are a zipped collection of resources and code known as an application package file (APK). Within these the virtual machine code to be executed exist in a classes.dex file. A memory optimized, de-duplicated collection of Java class files that have been collected into a single file for use on less robust devices. These files are very similar to, and can be converted to jar files.

In recent years there has been increased interest in the JVM and creating new languages to run on it or porting old ones. Jython, JRuby, Scala and Clojure being some of the more popular ones. This allows us to use the run-anywhere nature of the JVM without writing our code in Java. Building on this we can, with a few modifications, load the Dalvik bytecode into the JVM implementation of our choice and have access to all of the classes, methods, and information stored in it, along with the ability to execute this code in arbitrary ways.

Often when we create software for new platforms hard learned security lessons on previous platforms fade and we make the same mistakes all over again. This is a particularly common occurrence in the mobile space as its growth has not only led to a land-grab for space in app stores, but has also drawn many new and first time developers and an enormous focus on providing as "frictionless" of an experience as possible for its users. Some amazing things have come out of this, and the user experience provided by many mobile applications is second to none, however in some this has led to these apps being trusted too much by the web based APIs that they interact with.

For many current applications this trust comes by way of using the Oauth 1.0 protocol. Full details of the OAuth protocol can be found elsewhere, but briefly it allows for clients and their users to authenticate themselves to an API when making requests. This is accomplished by assigning a secret to the client application, which is used by itself, or in conjuction with a users application specific key, to generate a SHA1-HMAC signature of the request and its parameters. Originally OAuth was designed for server to server communications which allowed the client secret to remain, well secret, but has been widely adopted in mobile applications as well putting it in the hands of users everywhere and essentially rendering it useless for trust purposes. However, the opposite seems to be true and major vendors in the space are increasing using

this authentication as a way to restrict what apps have permission to use the service and what functions they can access. It can be thought of as similiar to Digital Rights Management solutions, in that a user may have an account with the service but unable to use it through an unapproved client.

We are able to build on publically available tools to expose this functionality and use it to perform tasks that would otherwise be privlidged, such as creating accounts without having to pass a captcha or bypassing throttling controls. We will demonstrate the latter behaviour and the steps needed to run the Dex code scripted in a JRuby instance. Following that we will generalize to code that will allow selection of a set of API endpoints and client secrets, iterating over them and comparing results to give a black box way of determining privileged client secrets.

Practicals:

APKs as libs:

As useful as this technique is for finding secrets without resorting to extensive reverse engineering, it is potentially more useful in the time savings past that. Often when interacting with an external service, especially one without documentation, much time is spent prototyping and duplicating existing client functionality. By scripting the APK, we can explore the functionality in a dynamic and often much quicker and less error prone way.

Translating and Gathering Contents:

First step to being able to load the code is to translated to a jar file.

```
$ unzip target.apk
$ d2j-dex2jar.sh classes.dex -o target.jar
```

Undestanding the code:

Again the best approach is to reach for established tools in the Java ecosystem. We can use the output from jd-gui to guide our understanding of the functionality provided by the (now) Jar file. Often times this will lead to a full listing of classes, methods, and their expected arguements, but other times we will find listings that have been obscured using Proguard, or a similar process. This serves as a detriment, but only means we will have to go deeper to understand and use the APK for our purposes.

Assuming that the classes are visible we can load them into our JRuby instance and begin exploring with the code similar to the following.

```
require 'java'
require 'target.jar'
```

```
require 'appropriate_android_version.jar'

java_import UserFactory

UserFactory.validate_user(<username>, <email_address>, <password>)
```

From there the APK is open for our use with the only limit being the scripting authors understanding. However if the code has been obsfucated more word is required. For this we can revert to somewhat standard practices of finding candidate methods that make API calls we're interested in using via grep or android debugging, or perhaps just extracting the secret keys and plugging it into existing libraries. A full overview of this topic is outside the scope of this document, however we will briefly describe a simple, and somewhat naive method, that is useful for instructive purposes and allows others to build on for their own workflows for those who are not experienced with reverse engineering, modifying, and rebuilding Android applications for debugging.

For reversing I recommend using a more direct approach than jd-gui, and disassembling the APK and its classes.dex file into the smali assembly language. This provides a much lower level representation of the code, as jd-gui provides a decompilation of the code which may not be completely accurate, smali is disassembly of the code which will be executed directly on the Dalvik virtual machine. A very nice tool for dissassembling the apk is the aptly name apktool.

```
$ apktool d target.apk output_directory
```

In the output directory we'll see that every class file has been translated into a smali file, willed with the assembly. Thankfully smali is quite straightforward, and much more pleasant to reverse than many other assembly languages. As in this example we're searching for Oauth keys a good place to start would be searching for strings related to cryptographic functions, say "hmacsha1." It is unlikely that the keys will be found in plain text, but we can do some dynamic exploration by modifying and rebuilding the APK. We can turn to the simplest form of debugging available, printf debugging, or in this case log debugging. By inserting the following lines into the smali code of the method we're targetting we can log parameters and variables local to the method.

```
 const-string v2, "First Parameter, P0"
invoke-static {v2, p0}, Landroid/util/Log;->d(Ljava/lang/String;Ljava/lang/String;)I
 const-string v2, "Second Parameter, P1"
invoke-static {v2, p0}, Landroid/util/Log;->d(Ljava/lang/String;Ljava/lang/String;)I
```

We can now rebuild our modified apk, again using apktool:

```
$ apktool b target_modified.apk
```

Loading this onto an android device and running the logcat program in an adb session we can

see the generated log messages, and with a bit of itteration soon find out keys.

Key Privilege Enumeration:

After performing key recovery an important, but often annyoing step is discovery the permissions it has relative to other keys for the same API. We include some code to help with this, building on the indispensable web testing platform Burpsuite we extend functionality to resend arbitrary captured requests appropriately signed from the collection of keys presented. Responses are collected and sent to the Comparer portion of Burpsuite for further analysis.

Caveats:
As the Dalvik virtual machine and the Java virtual machine do not perfectly match up, there may be subtle bugs in using this technique. At the time of this writing we have not encountered any problems that kept us from using these techniques for the purposes of key retrieval, creating and control and thousands of zombie social network accounts, and reusing/bypassing cryptographic features, but results for specific APKs may vary.

Concluding:

By applying the techniques above the reader should be able to script android packages for various purposes, and have a more complete understanding of some of the approaches to modern web API security and some of its pitfalls.