Penetration Testing Android Applications

Author:

Kunjan Shah
Security Consultant
Foundstone Professional Services

Table of Contents

Abstract

Mobile application penetration testing is an up and coming security testing need that has recently obtained more attention with the introduction of the Android, iPhone and iPad platforms among others. The mobile application market is expected to reach a size of $9 billion by the end of 2011[1] with the growing consumer demand for smartphone applications, including banking and trading. A plethora of companies are rushing to capture a piece of the pie by developing new applications, or porting old applications to work with the smartphones. These applications often deal with personally identifiable information (PII), credit card and other sensitive data.

This paper focuses specifically on helping security professionals understand the nuances of penetration testing on Android applications. It attempts to cover the key steps the reader would need to understand such as setting up the test environment, installing the emulator, configuring the proxy tool and decompiling applications etc. It also provides an introduction to security tools available for the Android platform. To be clear this paper does not attempt to discuss the security framework of the Android platform itself, identify flaws in the operating system, or try to cover the entire application penetration testing methodology.

[1] http://www.mgovworld.org/topstory/mobile-applications-market-to-reach-9-billion-by-2011

Background

Android is a Linux-based platform developed by Google and the Open Handset Alliance. Application programming for it is done exclusively in Java. The Android operating system software stack consists of Java applications running on a Dalvik virtual machine (DVK). The current version as of August 2010 is 2.2. There are over 90,000 applications available in the Android market.

Mobile phones these days are miniature computers and the applications that run on them are similar to web applications or thick client applications. Given this once you have a proxy setup and the code decompiled security testing is narrowed down to performing penetration testing or code review as you would on any other application.

Setting up the Test Environment

There are several ways to test mobile applications *e.g.*:

1. Using a regular web application penetration testing chain (browser, proxy).
2. Using WinWAP with a proxy[2].
3. Using a phone emulator with a proxy[3].
4. Using a phone to test and proxy outgoing phone data to a PC.

In this paper we will focus on using a phone emulator with a proxy as it is the easiest and cheapest option out there for testing mobile applications. For some platforms, this can be difficult but for Android applications, use of an emulator is easy and effective.

Requirements:

- Computer running a Microsoft Windows operating system
- Java 5 or 6
- Eclipse 3.5
- Android SDK 2.2
- Fiddler

[2] http://www.winwap.com/desktop_applications/winwap_for_windows
[3] http://speckyboy.com/2010/04/12/mobile-web-and-app-development-testing-and-emulation-tools/

Installing the Android SDK

The first step before any testing can commence is to download and install the Android SDK[4]. For the purposes of this paper, we will use Microsoft Windows for testing. Your computer needs to have Java 5 or 6 and Eclipse in order to install the SDK. The installation process is very easy on Microsoft Windows and is self explanatory - simply run setup.exe. Next, add the `SDK_ROOT` to system variables pointing to the /tools folder and add `%SDK_ROOT%` to the PATH variable as shown below.

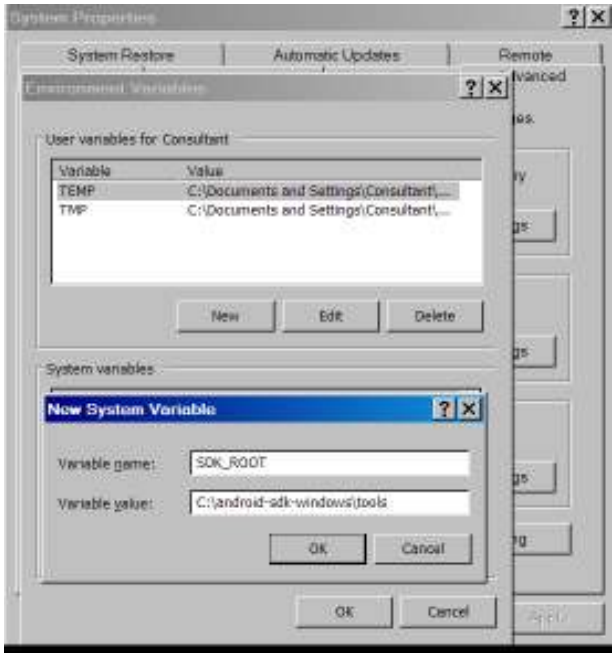

Figure 1: System variables to set to avoid specifying the whole path when running Android SDK commands

[4] http://developer.android.com/sdk/index.html

Starting the Emulator

The Android emulator comes packaged with the SDK. It is a QEMU-based device-emulation tool that you can use to design, debug, and test your applications in an actual Android run-time environment. Before starting the emulator you need to create an Android Virtual Device (AVD). Navigate to Eclipse > Window menu > Android SDK and AVD Manager > Virtual Devices and create a new AVD with the default settings.

To start the emulator, enter the following command: `emulator -avd testavd`. We will look at more advanced options that you can specify with this command later in the paper. It will launch the emulator as shown in the screenshot below.

Figure 2: Basic command to launch the emulator

Figure 3: The Android emulator in action

Next, download any Android application or create one of your own using the "App Inventor" to test with the emulator and other tools mentioned in this paper.

How to Install and, Uninstall Android Applications on the Emulator

You need to obtain an application's ".apk" (Android Package) file in order for you to perform penetration testing. Use the Android Debug Bridge (ADB) that comes with the SDK to install the files into the emulator.

- Open a command prompt and enter the following command to install any Android Package file

```
adb install <path of the .apk file>
```

```
C:\android-sdk-windows\tools>adb install C:\SyncClientBinary.apk
1144 KB/s (0 bytes in 750906.000s)
        pkg: /data/local/tmp/SyncClientBinary.apk
Success

C:\android-sdk-windows\tools>_
```

Figure 4: Installing Android applications to the emulator

Figure 5: Newly installed application in the emulator

- If you get an error message during the installation, try the following commands:

```
adb kill-server
adb start-server
```

- If the install fails due to size constraints, restart the emulator by executing the following command

```
emulator -partition-size 256 -memory 512 -avd testavd
```

```
C:\Documents and Settings\Consultant>emulator -partition-size 256 -memory 512 -a
vd testavd
```

Figure 6: Starting the emulator with additional space and memory

- You can uninstall the application either using the command prompt or the emulator. To use the command prompt open the "adb shell", navigate to the "app" folder and use the `rm` command to delete the ".apk" file as shown below.

```
C:\android-sdk-windows\tools>adb shell
# cd data
cd data
# cd app
cd app
# ls
ls
com.funambol.android-1.apk
# rm com.funambol.android-1.apk
rm com.funambol.android-1.apk
rm failed for com.funambol.android-1.apk, No such file or directory
# rm com.funambol.android-1.apk
rm com.funambol.android-1.apk
```

Figure 7: Uninstalling an application from the emulator

- Alternatively, to uninstall the application using the emulator, navigate to Menu > Settings > Applications > Manage Applications, select the application and press uninstall as shown below.

Figure 8: Uninstalling an application using the emulator

Setting up a Proxy Tool

If the application is using HTTP(s), or is a website that you are testing on the Android browser, the next step is to setup a proxy tool such as Fiddler or Paros. There are 4 main ways of setting up such a proxy:

1. Specify the proxy details when starting the emulator using the command below. This command is to use a proxy listening on port 8888 (the default configuration for Fiddler). If you are using any other proxy port (e.g. port 8080 for Paros) then change the port number.

```
emulator -avd testavd -http-proxy http://localhost:8888
```

```
C:\android-sdk-windows\tools>emulator -avd testavd -http-proxy http://localhost:
8888
```

Figure 9: Command to setup a web proxy with the emulator

2. The second option is to specify the proxy details in the emulator APN settings as shown below. Navigate to Home > Menu > Wireless & Networks > Mobile Networks > Access Point Names. Update the following settings:

   - **Name**: Internet
   - **APN**: Internet
   - **Proxy**: IP address of your computer *e.g.* 192.168.1.3
   - **Username**: <Not Set>
   - **Password**: <Not Set>

Figure 10: Setting up a proxy tool using the APN settings of the emulator

3. The third option is to specify it using the adb shell using the export command to set an environment variable, for example:

```
export HTTP_PROXY=http://localhost:8888
```

```
C:\Documents and Settings\Consultant>adb shell
# export HTTP_PROXY=http://localhost:8888
export HTTP_PROXY=http://localhost:8888
#
```

Figure 11: Command for setting up a proxy using the adb shell

4. The final alternative is by changing the proxy settings in the settings database from where the android web browser reads. The settings database uses SQLite. Familiarity with basic SQL commands is recommended if you plan to use this method. Change the hostname and port information appropriately is illustrated in the command below leaving everything else as is.

```
> adb shell
# sqlite3
/data/data/com.google.android.providers.settings/databases/settings.db
sqlite> INSERT INTO system VALUES(99,'http_proxy','localhost:8888');
sqlite>.exit
```

Once you have used one of these options your proxy should start seeing requests and responses. The figure below shows Fiddler intercepting HTTP requests sent by the emulator browser. Having a web proxy intercepting requests is a key piece of the puzzle. From this point forward, penetration testing is similar to that of regular web applications.

Figure 12: Fiddler intercepting requests sent by the emulator browser

Android Application Penetration Testing Toolkit

The Android SDK comes with several utilities that, although not designed specifically for security testing, could come in handy for penetration testing. In addition, there are several tools out there such as the Manifest explorer, Intent Sniffer and Intent Fuzzer that you could use as part of your toolkit as well.

Before we look at specific tools it helps to point out a useful tip when testing web applications on the Android platform - leverage the hidden debug menu. In order to get access to this menu follow the steps below:

- Navigate to the Android browser in the emulator
- Enter `about:debug` in the address bar and click →
- Go to Menu → More → Settings
- Scroll down to the bottom to see the now enabled debug menu
- The "`UAString`" setting lets you change the User Agent string of the browser when in this menu. Similarly, there are other settings that you can put to good use during penetration testing.

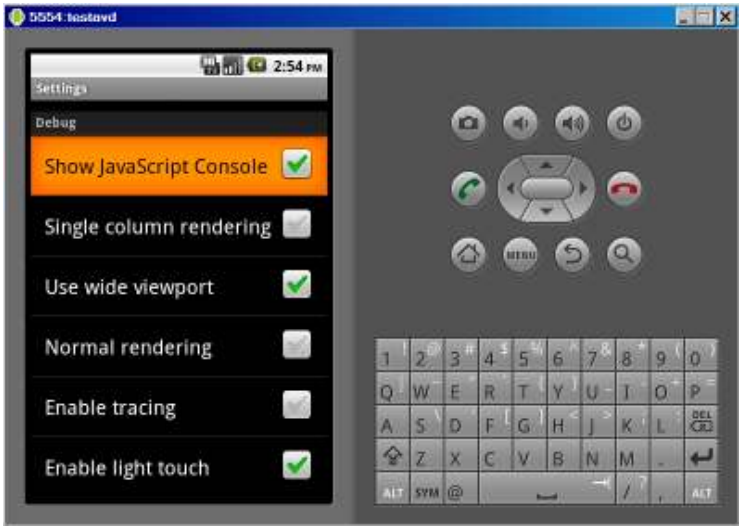

Figure 13: Debug menu

Android Debug Bridge (ADB)

We have already seen this tool in action when installing Android applications. It is part of the Android SDK. It has its own shell, which allows you to execute Linux commands such as `ls -l`. The Android Developer's Guide[5] lists the full range of ADB shell commands but we highlight a few below.

- ADB could be used to locate all the emulators and Android devices connected to the computer using the command below:

  ```
  adb devices
  ```

  

  Figure 14: Finding emulators and devices on a given computer.

  In our case the command found one instance of the emulator running. If multiple instances are running you can use the `-s` option in order to run commands against a specific device or emulator.

  ```
  adb -s emulator-5554 install Foobar.apk
  ```

- Another important command provided by the ADB is to pull/push files to and from the emulator/device instance's data file. This could be useful if you want to download files from the emulator/device to your computer and review or process them. We will examine this functionality in more detail when we discuss the decompilation process.

- The `dumpsys` or `dumpstate` commands can be used to dump system data to the screen or a file as shown below. This file could contain important security related information. Alternatively you could use the Dalvik Debug Monitor Service (DDMS) for this purpose.

[5] http://developer.android.com/guide/developing/tools/adb.html

```
------ MEMORY INFO (/proc/meminfo) ------
MemTotal:           94096 kB
MemFree:             2212 kB
Buffers:                0 kB
Cached:             36184 kB
SwapCached:             0 kB
Active:             38360 kB
Inactive:           41472 kB
Active(anon):       23940 kB
Inactive(anon):     24160 kB
Active(file):       14420 kB
Inactive(file):     17312 kB
Unevictable:          280 kB
Mlocked:                0 kB
SwapTotal:              0 kB
SwapFree:               0 kB
Dirty:                  0 kB
Writeback:              0 kB
AnonPages:          43948 kB
Mapped:             30520 kB
Slab:                3676 kB
SReclaimable:        1204 kB
SUnreclaim:          2472 kB
PageTables:          5068 kB
NFS_Unstable:           0 kB
Bounce:                 0 kB
WritebackTmp:           0 kB
CommitLimit:        47048 kB
Committed_AS:     1491976 kB
VmallocTotal:      876544 kB
VmallocUsed:        15456 kB
VmallocChunk:      858116 kB
```

Figure 15: Cropped output of the dumpsys command.

```
  PID   TID CPU% S     VSS     RSS PCY UID      Thread          Proc
  327   327  27% R    964K    440K  fg shell    top             top
   57    58   3% S 160140K  31776K  fg system   HeapWorker      system_server
   57   105   1% S 160140K  31776K  fg system   er$SensorThread system_server
   57    66   1% S 160140K  31776K  fg system   er.ServerThread system_server
   57    63   0% S 160140K  31776K  fg system   SurfaceFlinger  system_server
   35    35   0% S   1616K    404K  fg keystore keystore        /system/bin/keys
tore
   36    36   0% S    740K    328K  fg root     init.goldfish.s /system/bin/sh
   37    37   0% S    848K    356K  fg root     qemud           /system/bin/qemu
d
   39    39   0% S   3384K    176K  fg root     adbd            /sbin/adbd
   39    41   0% S   3384K    176K  fg root     adbd            /sbin/adbd
   39    84   0% S   3384K    176K  fg root     adbd            /sbin/adbd
   39    85   0% S   3384K    176K  fg root     adbd            /sbin/adbd
   50    50   0% S    792K    276K  fg root     qemu-props      /system/bin/qemu
-props
   57    57   0% S 160140K  31776K  fg system   system_server   system_server
   57    59   0% S 160140K  31776K  fg system   Signal Catcher  system_server
   57    60   0% S 160140K  31776K  fg system   JDWP            system_server
   57    61   0% S 160140K  31776K  fg system   Binder Thread # system_server
   57    62   0% S 160140K  31776K  fg system   Binder Thread # system_server
   57    65   0% S 160140K  31776K  fg system   DisplayEventThr system_server
   57    68   0% S 160140K  31776K  fg system   ActivityManager system_server
   57    71   0% S 160140K  31776K  fg system   ProcessStats    system_server
   57    72   0% S 160140K  31776K  bg system   PackageManager  system_server
   57    73   0% S 160140K  31776K  fg system   FileObserver    system_server
   57    74   0% S 160140K  31776K  fg system   AccountManagerS system_server
   57    76   0% S 160140K  31776K  bg system   SyncHandlerThre system_server
   57    77   0% S 160140K  31776K  fg system   UEventObserver  system_server
   57    78   0% S 160140K  31776K  fg system   PowerManagerSer system_server
   57    79   0% S 160140K  31776K  fg system   AlarmManager    system_server
   57    80   0% S 160140K  31776K  fg system   WindowManager   system_server
   57    81   0% S 160140K  31776K  fg system   InputDeviceRead system_server
[top: 1.7s elapsed]
```

Figure 16: Cropped version of the dumpstate command.

MKSDCARD

The `MKSDCARD` command allows you to create a virtual SD card for the emulator, by creating a FAT32 disk image. It is possible that the application you are testing requires an SD card to install a database or other files. This is therefore a useful utility when you want to test the application and are using an emulator instead of a physical device.

- Use the `mksdcard` command to create a virtual SD card.

  ```
  mksdcard [-l label] <size>[K|M] <file>
  ```

- Now, execute the `-sdcard` option to start the emulator by specifying the location of the SD card file.

  ```
  emulator -sdcard <file specified in the command above>
  ```

You may find hidden secrets by parsing through the files stored on the SD card by the application. Always be in the lookout for passwords, PINs, PII, and other sensitive information.

SQLITE3

From the ADB shell you can also run the `sqlite3` command line program to query databases created by Android applications and stored in the device memory. These also may reveal sensitive information such as are passwords or PINs hashed or stored in clear text. Such databases are stored with a ".db" file extension.

- Navigate to /data/data/<application>/databases/<nameofthedatabase>.db

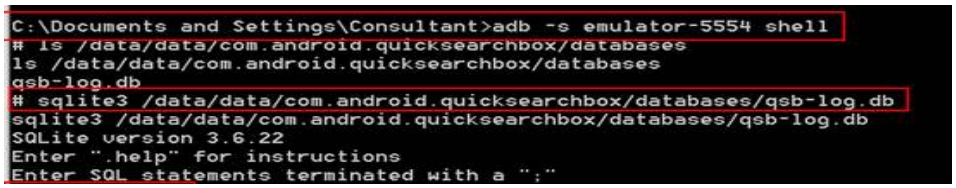

Figure 17: Navigating to the database file stored on the emulator.

- Execute the `.table` command to list all the tables and `.schema <tablename>` to list the structure of the table as shown below.

```
.exit
# sqlite3 /data/data/com.android.quicksearchbox/databases/qsb-log.db
sqlite3 /data/data/com.android.quicksearchbox/databases/qsb-log.db
SQLite version 3.6.22
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .table
.table
android_metadata  clicklog          shortcuts         sourcetotals
sqlite> .tables
.tables
android_metadata  clicklog          shortcuts         sourcetotals
sqlite> .schema android_metadata
.schema android_metadata
CREATE TABLE android_metadata (locale TEXT);
sqlite>
```

Figure 18: Output of .table and .schema commands.

- You can also execute SQL commands like `select * from shortcuts;`

Manifest Explorer

Every application running on Android has an `AndroidManifest.xml` file. This file is very important from a security perspective as it defines the permissions an application requests. The Manifest Explorer tool[6] is a utility that allows you to review this XML file with ease. When testing it is important to verify that the application follows the principle of "least privilege" and does not use permissions that are not required for it to function.

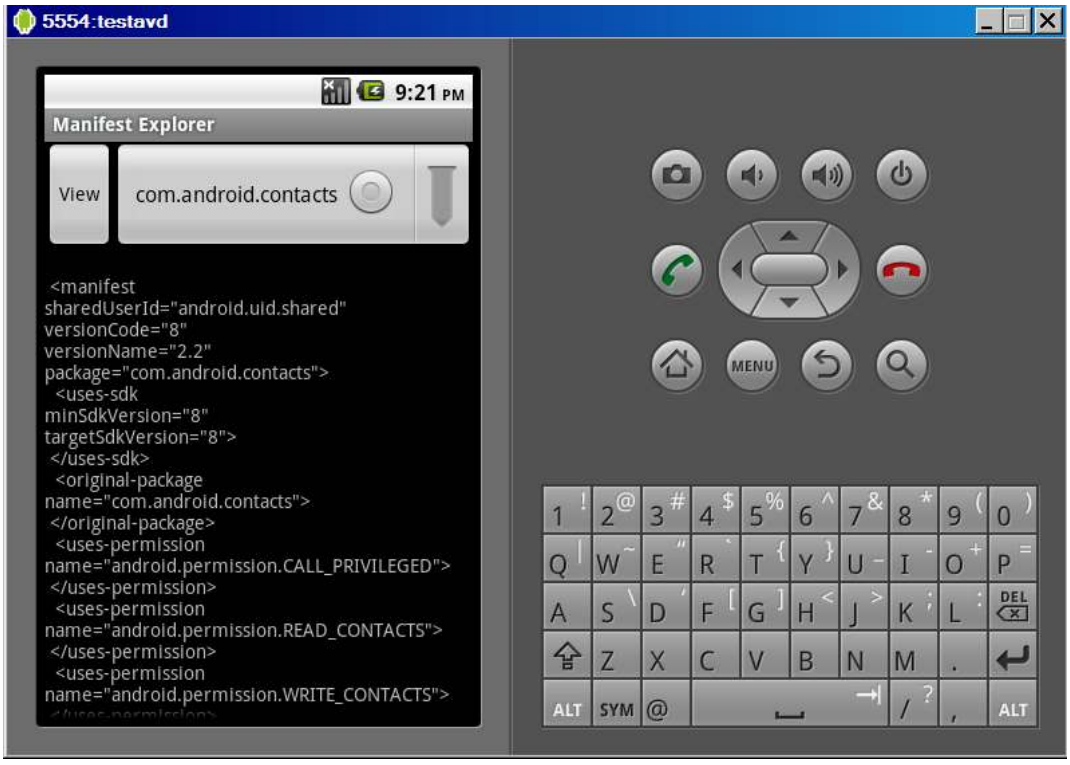

Figure 19: Manifest Explorer

[6] https://www.isecpartners.com/files/ManifestExplorer.zip

Intent Sniffer

Intent is a mechanism in Android to move data between processes. It forms the core of Android's Inter Process Communication (IPC). Intents could indicate a number of actions such as startservice, sendbroadcast etc. The Intent Sniffer tool[7] performs monitoring of Intents.

PROCRANK

The `procrank`[8] command shows the listing of processes running on the Android device as shown below. This is similar to the `ps` command but, adds additional columns such as Vss (indicates how much virtual memory is associated with each process) and Pss (Pss is Rss reduced by a percentage according to how many processes share the physical pages.).

```
# procrank
procrank

  PID      Vss      Rss      Pss      Uss  cmdline
   59   30132K   29532K   14286K   12004K  system_server
  152   24272K   24272K    8668K    6532K  android.process.acore
   32   23276K   23276K    7420K    5000K  zygote
  114   21780K   21780K    6958K    5196K  com.android.launcher
  113   20592K   20592K    5832K    3876K  com.android.phone
  108   18100K   18100K    4167K    2928K  jp.co.omronsoft.openwnn
  222   18072K   18072K    3952K    2612K  com.android.email
  205   17632K   17632K    3556K    2164K  com.android.mms
  194   17128K   17128K    3366K    2164K  android.process.media
  117   16984K   16984K    3086K    1560K  com.android.settings
  177   16740K   16740K    3041K    1708K  com.android.quicksearchbox
  158   16804K   16804K    2834K    1592K  com.android.alarmclock
  170   16268K   16268K    2575K    1336K  com.android.music
  188   15916K   15916K    2476K    1344K  com.android.protips
   33    1580K    1580K     685K     584K  /system/bin/mediaserver
  248     524K     524K     320K     312K  procrank
   31     544K     544K     218K     200K  /system/bin/rild
    1     204K     204K     185K     184K  /init
   29     400K     400K     166K     156K  /system/bin/netd
   39     176K     176K     160K     160K  /sbin/adbd
   28     384K     384K     153K     144K  /system/bin/vold
   37     344K     344K     121K     112K  /system/bin/qemud
  234     332K     332K     103K      76K  /system/bin/sh
   36     316K     316K      87K      60K  /system/bin/sh
   26     316K     316K      87K      60K  /system/bin/sh
   35     280K     280K      84K      76K  /system/bin/keystore
   51     312K     312K      82K      72K  /system/bin/qemu-props
   34     300K     300K      81K      72K  /system/bin/installd
   27     264K     264K      79K      72K  /system/bin/servicemanager
   30     240K     240K      59K      52K  /system/bin/debuggerd
```

Figure 20: Output of the Procrank utility

STRACE

`Strace`[9] is a debugging tool that traces system calls and signals. This utility comes installed with the Android SDK. It is very useful when testing an application that is not easy to intercept using Fiddler or other HTTP proxy tools. Just specify the process ID of the application which in turn can be discovered using the `Procrank` command described above.

[7] https://www.isecpartners.com/files/IntentSniffer.zip

[8] http://elinux.org/Android_Memory_Usage#procrank

[9] http://elinux.org/Android_Tools#strace

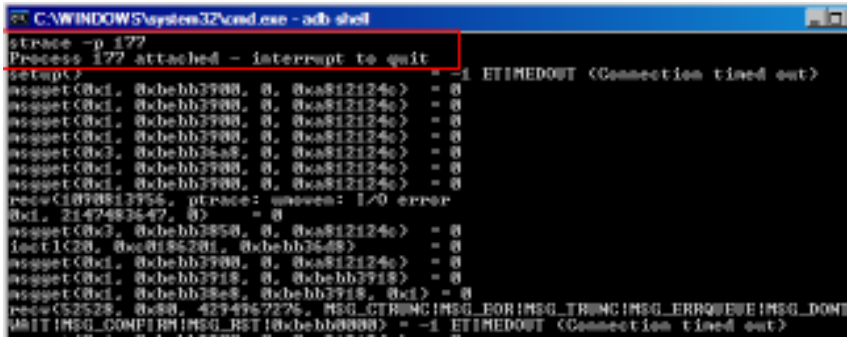

Figure 21: Output of the Strace utility

BUSYBOX

Busybox provides some Linux commands[10] that could be useful during the penetration testing process. It extends the capability of Android's toolbox. In order to install Busybox you can go through the following steps:

- Download the Busybox binary[11].
- Create a folder `/data/busybox`.
- Push the binary to this folder.
- Ensure the correct permissions are set using `Chmod 755 ./busybox`
- Install the application `./busybox –install`

To then execute a command within Busybox, navigate to the busy box directory and enter the following command

`./<command>` as shown below:

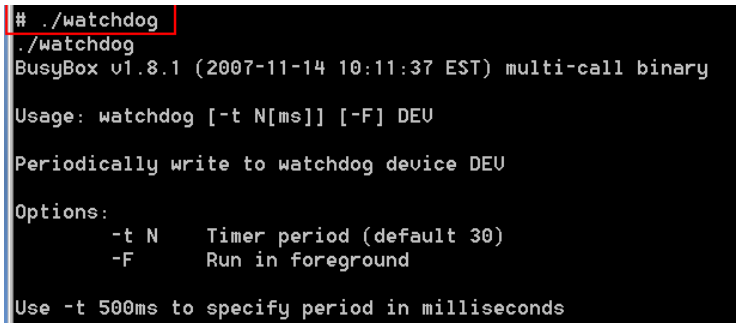

Figure 22: Executing the watchdog command that comes with Busybox.

[10] http://www.busybox.net/downloads/BusyBox.html
[11] http://benno.id.au/blog/2007/11/14/android-busybox

Decompiling Android Applications

- Android packages (".apk" files) are actually simply ZIP files. They contain the `AndroidManifest.xml`, `classes.dex`, `resources.arsc`, among other components. You can rename the extension and open it with a ZIP utility such as WinZip to view its contents.

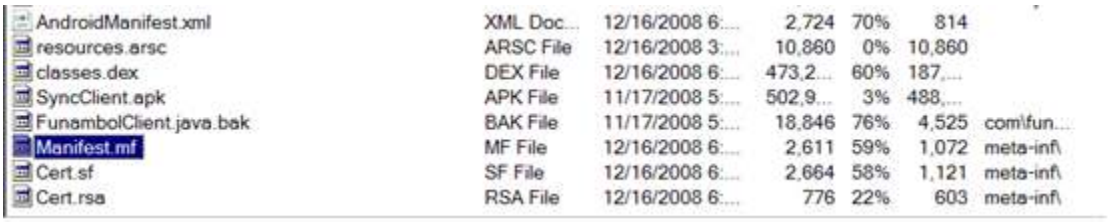

Figure 23: The contents of an .apk file

- It's most practical to transfer the ".dex" files to the computer in order to decompile them. The `classes.dex` files of the installed applications are located under `/data/Dalvik-cache`.

```
C:\Documents and Settings\Consultant>adb shell
# cd /data/dalvik-cache
cd /data/dalvik-cache
# ls
ls
system@framework@core.jar@classes.dex
system@framework@ext.jar@classes.dex
system@framework@framework.jar@classes.dex
system@framework@android.policy.jar@classes.dex
system@framework@services.jar@classes.dex
```

Figure 24: The default location for .dex files

- The ".dex" extension represents the Davlik executable format. In order to pull the `.class` files from it use the `dexdump` utility provided with the SDK. Use the following command to dump the .class file.

```
# dexdump -d -f -h data@app@com.funambol.android-2.apk@classes.dex >> sync.apk.d
ump
```

Figure 25: Command to dump the .dex files into byte code format

- Now, use the pull command to get it to a directory of the underlying computer as shown below.

```
C:\Documents and Settings\Consultant>adb pull /data/dalvik-cache/sync.apk.dum
:\sync.apk.dump
1529 KB/s (0 bytes in 7074192.004s)
```

Figure 26: Command to pull the .dump file to the computer

- The resulting dump file looks as shown in the figure below. If you are good at reading the Davlik byte code instructions, this is a good enough solution for you. But, people who are much more comfortable with Java could use the other options mentioned below to get a better output in Java like pseudo code.

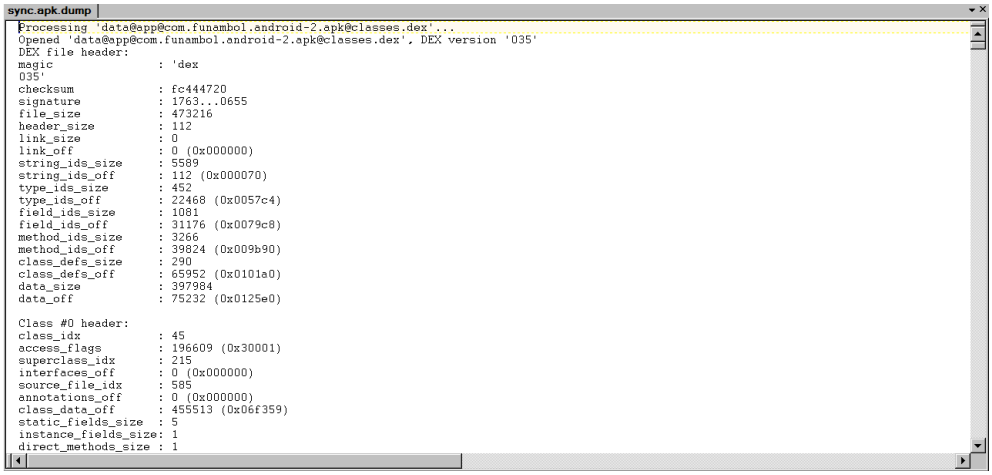

Figure 27: .dump file.

- You could also use the baksmali decompiler[12] which provides a much better output. To do this pull the .dex files onto the computer and then run the following command:

```
C:\>java -jar baksmali-1.2.3.jar -o C:\decompileddex C:\SyncClientBinary\classes
.dex
```

Figure 28: Command to decompile .dex files into Java source code

- The output of the decompiled .dex file is shown below. As you will notice it is much more readable to most people than the Davlik byte code. Based on our research there currently is no way to get compatible Java code from a .dex file and we believe this is the best option available. Another alternative tool that does a similar job is the dedexer[13].

[12] http://code.google.com/p/smali/downloads/detail?name=baksmali-1.2.3.jar
[13] http://dedexer.sourceforge.net/

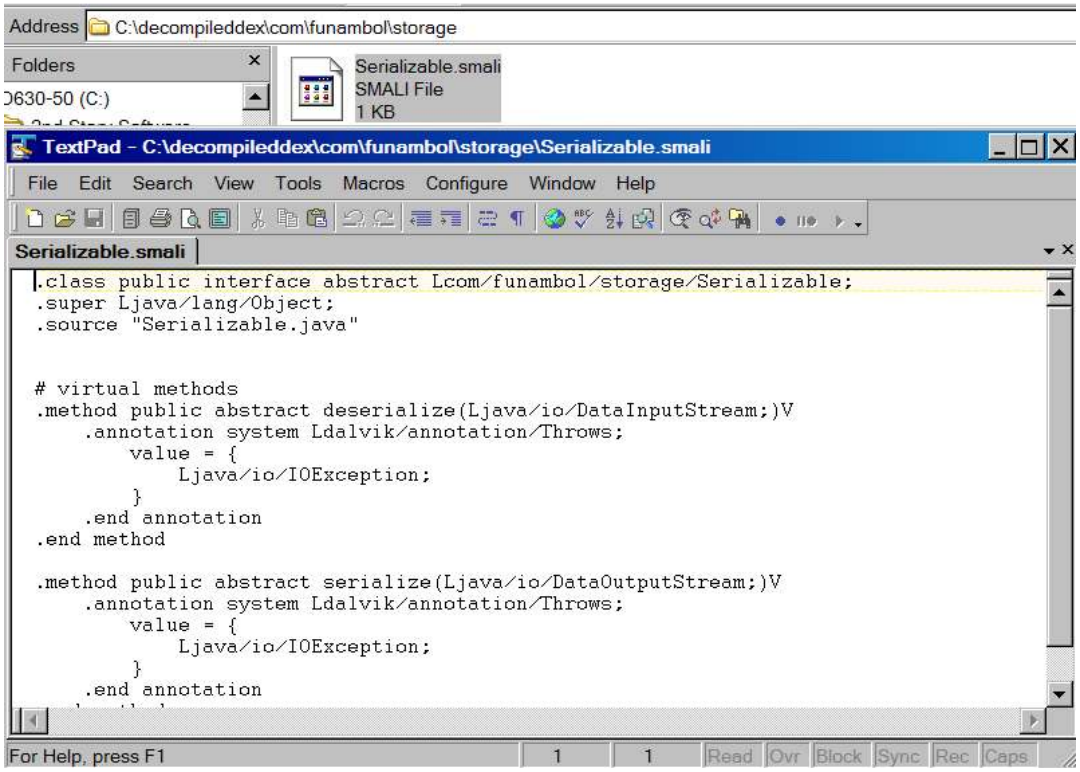

Figure 29: Output after decompiling the .dex file using the baksmali decompiler

File Permissions in Android

Android file permissions use the same model as Linux. To check the permissions of a file, go to the ADB shell and type `ls -l`. Every `.apk` file installed on the emulator has its own unique user ID. This prevents one application from accessing other application's data. Any file created by the application will be assigned that application's user ID, and will not normally accessible to other applications. However, if a new file is created with the `getSharedPreferences()`, `openFileOutput()`, or `createDatabase()` APIs you can specify the `MODE_WORLD_WRITEABLE` and `MODE_WORLD_READABLE` flags to allow other packages to read/write to this file globally. This presents a warning flag when performing a source code review. Consider therefore searching for the `MODE_WORLD_WRITEABLE` and `MODE_WORLD_READABLE` strings in the code, and question whether these are actually needed. It should be noted that such a check is only possible if you have access to the source code of the application since these flags will not show up in the decompiled code.

About the Author

Kunjan Shah is a Security Consultant at Foundstone Professional Services, A division of McAfee based out of the New York office. Kunjan has over 5 years of experience in information security. He has dual Master's degree in Information Technology and Information Security. Kunjan has also completed certificates such as CISSP, CEH, and CCNA. Before joining Foundstone Kunjan worked for Cigital. At Foundstone Kunjan focuses on web application penetration testing, thick client testing, mobile application testing, web services testing, code review, threat modeling, risk assessment, physical security assessment, policy development, external network penetration testing and other service lines.

Acknowledgements

I would like to thank Rudolph Araujo, Jeremiah Blatz and Christopher Silvers for reviewing this paper and providing useful feedback, and suggestions on making it better.

About Foundstone Professional Services

Foundstone® Professional Services, a division of McAfee. Inc. offers expert services and education to help organizations continuously and measurably protect their most important assets from the most critical threats. Through a strategic approach to security, Foundstone identifies and implements the right balance of technology, people, and process to manage digital risk and leverage security investments more effectively. The company’s professional services team consists of recognized security experts and authors with broad security experience with multinational corporations, the public sector, and the US military.