AppUse - Android Pentest Platform Unified Standalone Environment

AppUse is designed to be a **weaponized environment for android application penetration testing.** It is a unique, **free** and rich platform aimed for mobile application security testing in the android environment.

https://www.appsec-labs.com
info@AppSec-Labs.com
Last updated April 2013

Contents

AppUse - Overview

AppUse (Android Pentest Platform Unified Standalone Environment) is designed to be a weaponized environment for Android application penetration testing. It is an OS for Android application pentesters - containing a custom Android ROM loaded with hooks which were placed at the right places inside the runtime for easy application control, observation and manipulation.

The heart of AppUse is a custom "hostile" Android ROM, specially built for application security testing containing a modified runtime environment running on top of a customized emulator. Using rootkit-like techniques, many hooks were injected into the core of its execution engine so that application can be easily manipulated and observed using its command & control counterpart called "ReFrameworker" (figure 1).

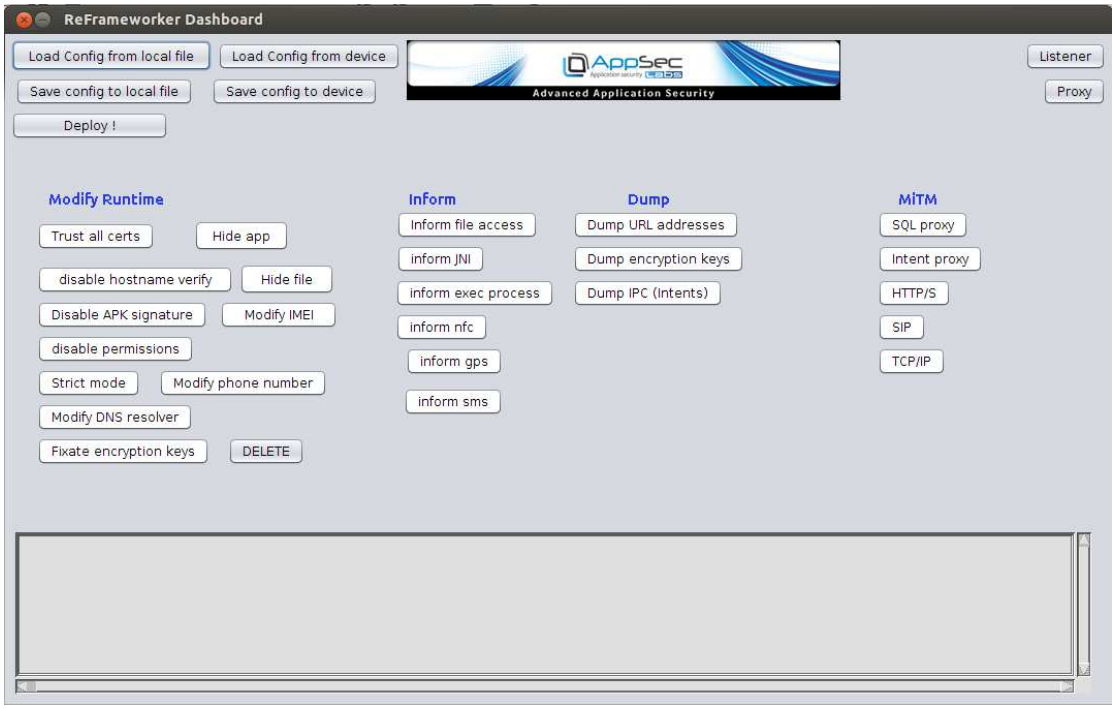


Figure 1 – the ReFrameworker dashboard

Other than this, AppUse also comes with everything the pentester needs in order to run and test target applications – the Android emulator , development tools , the required SDKs, decompilers, disassemblers, etc.

The AppUse environment is designed to be intuitive and productive as possible to the Android common pentesters and security researchers. It comes with the AppUse dashboard - an easy to use UI from which the user can control the whole environment. From installing an APK to the device, decompiling it, debugging it, manipulating its binaries and such – everything can be accomplished by clicking on a few buttons that take care of all the required steps and focusing on the really important matters (figure 2).

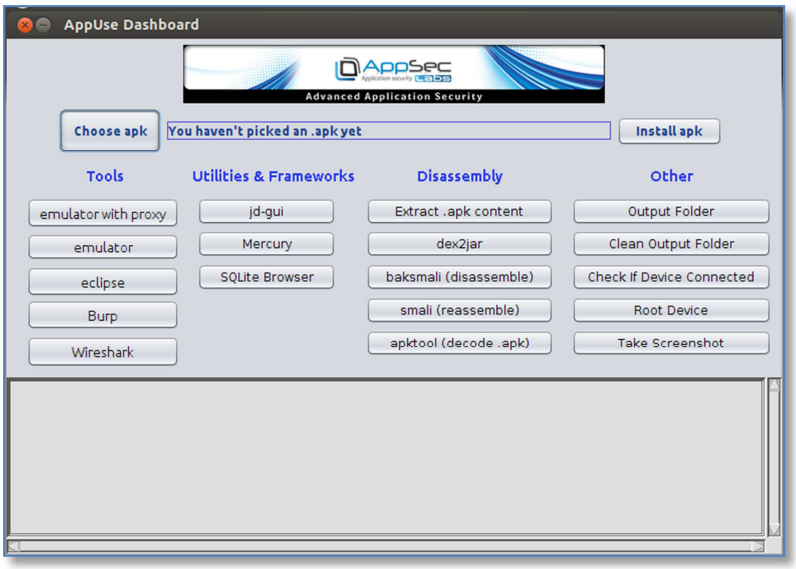


Figure 2 – the AppUse dashboard

Besides that, there are many Android "hack me" applications pre-installed on the AppUse environment, along with their server-side services. Having such target applications is really handy for the pentester when the need arises for testing a new tool or technique and some target is required to be around.

So to summarize, AppUse is combined of:

- Custom "hostile" Android ROM loaded with hooks
- ReFrameworker Android runtime manipulator
- Android Emulator
- Development tools for Android
- Hacking & reversing tools for Android
- Vulnerable applications
- The AppUse dashboard

AppUse can be downloaded here:

https://appsec-labs.com/AppUse

Runtime and OS

The AppUse OS is based on Linux Ubuntu which had been perfectly suited with common attacking tools embedded that can save time and increase efficiency.

Credentials:

Although AppUse will automatically log you in to root, you may find this data useful while interacting with the system.

Type	Username	Password
Operating System	Root	1
Emulator	[none]	1234

Terminal and Command Line Tools

The environment variables had been suited to the Android security researcher. Useful tools used in research, such as ADB, were preinstalled and configured on the machine. Moreover, all the research and attacking tools had been embedded in the PATH environment variable so they are accessible on any location on the terminal and have the tab autocomplete feature (figure 3,4)

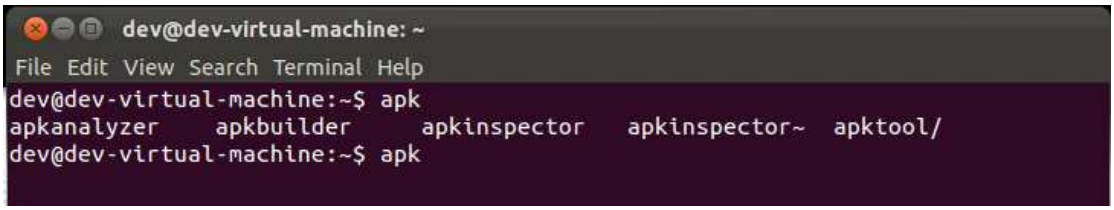


Figure 3 - Autocomplete feature – entering apk[tab] on the console.

Figure 4 - Autocomplete feature – entering ad[tab] on the console.

Development Tools

The AppUse environment includes Android development and debugging tools which can come in handy while performing applicative penetration testing. The environment has a preinstalled version of eclipse and ADT (figure 5) that can be used to write applications or exploits to cross-application vulnerabilities. Moreover, the environment has a preinstalled version of the iPython console that will ease the development of scripts and help testing special scenarios.

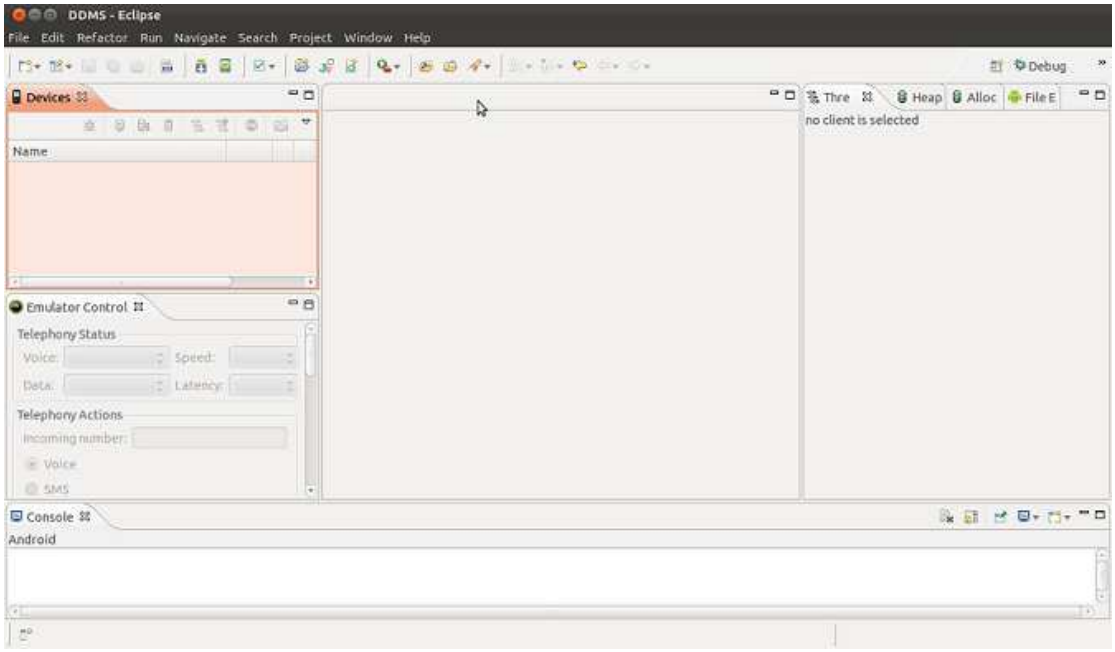


Figure 5 - Eclipse with ADT. Useful in writing cross-application vulnerabilities exploits.

AppUse Dashboard

The dashboard is the heart of AppUse testing environment. The dashboard is a GUI which organizes the testing tool and runtime environment that will be used in the research. The dashboard will put the puzzle together by linking all the data from all different tools and will save precious time in its special functionalities that will concatenate several actions together, as will be demonstrated further in this document.

To launch the dashboard, double click the **Launch Dashboard** link on your desktop, and immediately the dashboard will be launched (figure 6):

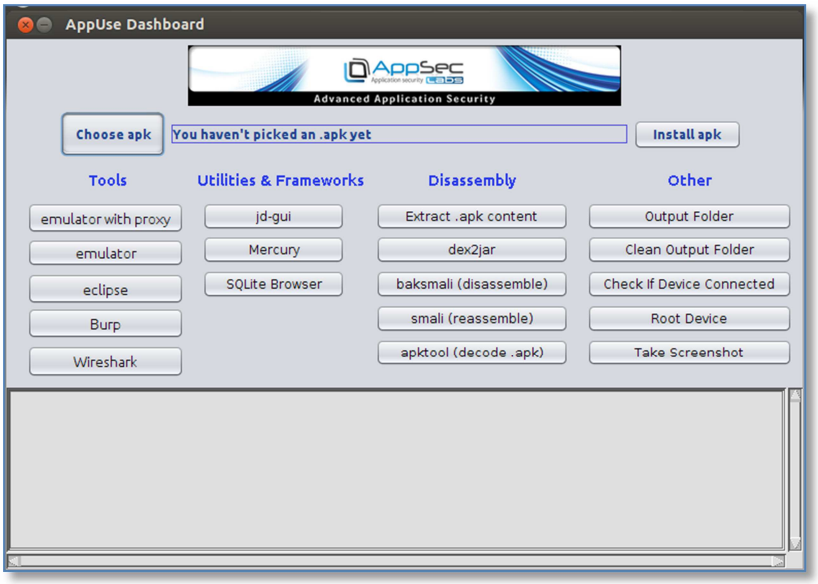


Figure 6 – launching the dashboard

Working with APK's

In the dashboard, the APK is the king. AppUse dashboard's goal is to enable researchers to start working with one-click actions. In order to achieve the goal, the dashboard is designed to operate on an APK and use it while invoking its other actions.

Choose APK

Choosing the APK is the most basic action the dashboard can do. Choose APK button will load an APK into the dashboard (figure 7) from the file system so that the dashboard will perform its actions with it. When starting a research on an application, this is the very first action that needs to be done.

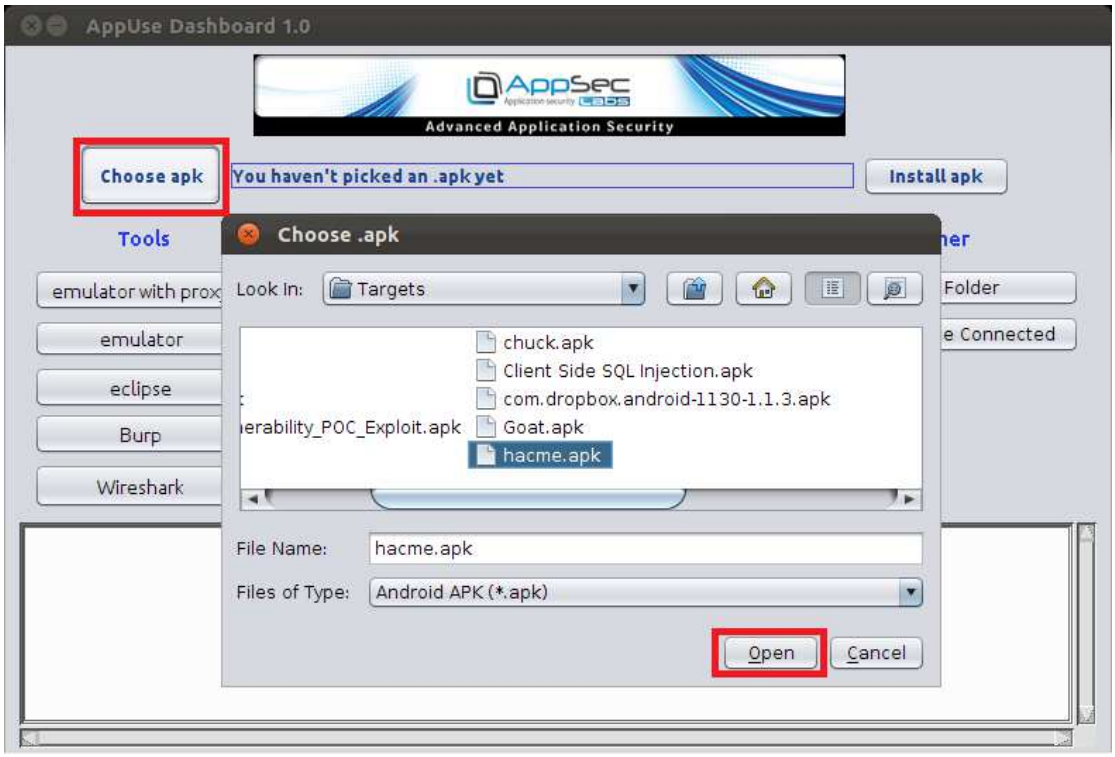


Figure 7 - Choose APK action. Example APK's can be found under the Targets folder. The Targets folder is discussed

Installing APK

The Install APK button (figure 8) will install an APK on a running emulator invoked from the Dashboard.

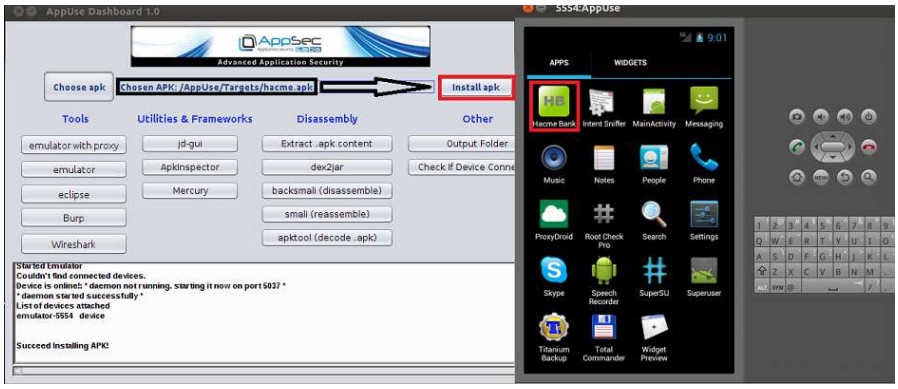


Figure 8 - Installing an APK on the emulator via a click of a button

Rooting the emulator

Root privileges on an emulator may come in handy in a penetration test but normally can consume time. AppUse Dashboard has a built in option to automatically root the emulator with a click of a button (figure 9) and later on verify it is rooted (figure 10).

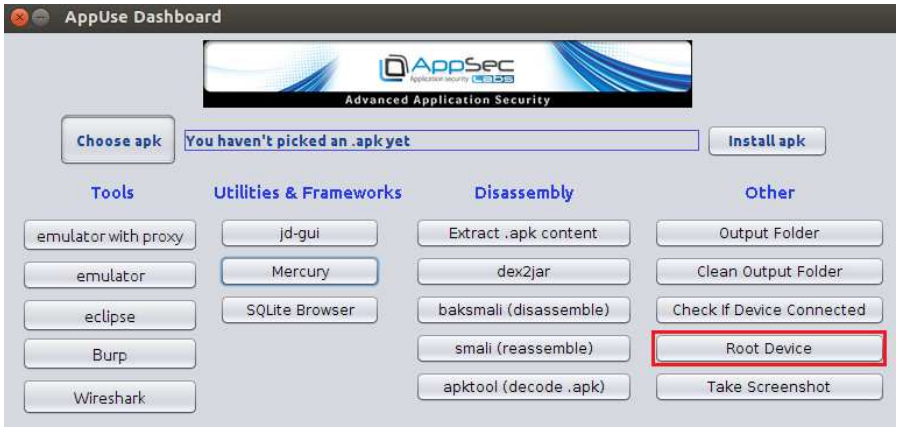


Figure 9 - The Root Device button – will root the emulator with a click.

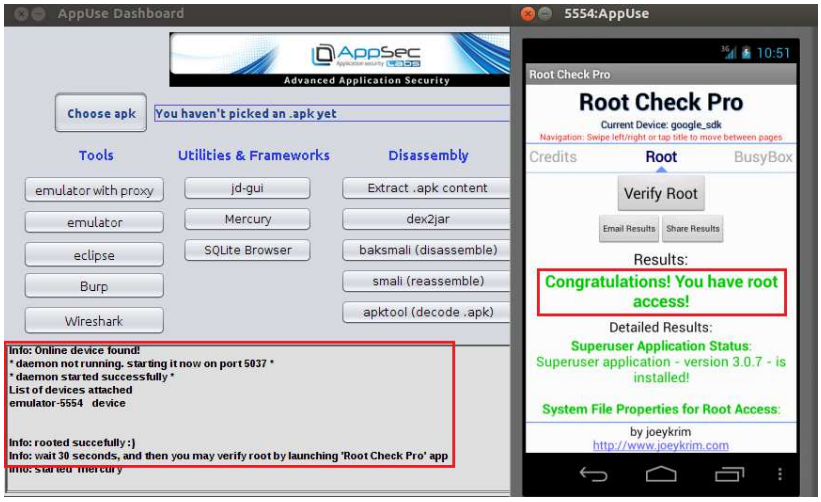


Figure 10 - The results from Root Check Pro application – successfully rooted the emulator with a click.

The output folder

The main goal for AppUse is to organize all the pentester's work. In order to accomplish that, all of the work in AppUse on an application will be saved into one directory, the Output folder (figure 11). The Output directory (will be elaborated later on this document) contains all the output from all the tools in the dashboard. When you open an APK via APKTool, its output is shown in the Output folder. When you convert a .dex file to .jar, the classes.dex in the output folder will be chosen automatically and in a click of a button you will convert the APK's .dex to a JAR.

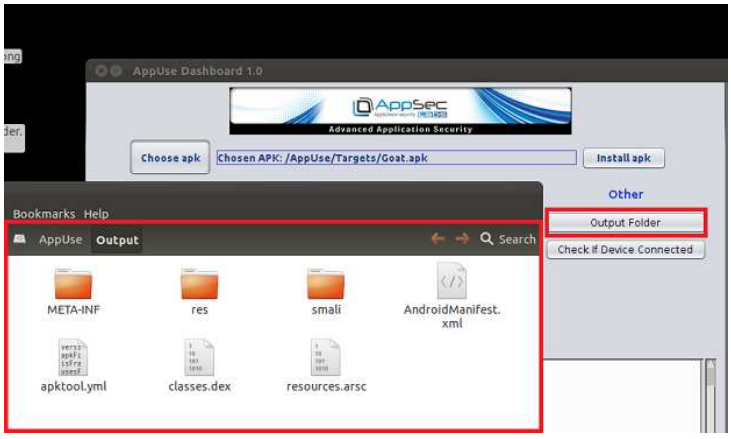


Figure 11 - Output folder – the Dashboard's workbench.

Device Connectivity

AppUse has a support for external devices. You may either work with the emulator or plug in a real Android device and have it interacting with AppUse (figure 12).

Figure 12- Checking device connectivity via the AppUse Dashboard

Runtime Modifications and Inspection via AppSec ReFrameworker

The emulator in AppUse is modified to suit the needs of the pentester. It comes with a premade ROM that has modifications to the Dalvik runtime and has preinstalled tools to interact with the system.

The heart of AppUse is a custom "hostile" Android ROM, specially built for application security testing containing a modified runtime environment running on top of a customized emulator. Using a rootkit like techniques, many hooks were injected into the core of its execution engine so that application can be easily manipulated and observed using its command & control counterpart called "ReFrameworker".

How it works – an overview

The Android runtime was compiled with many hooks placed into key placed inside its code. The hooks look for a file called "Reframeworker.xml", located inside /data/system. So each time an application is executed, whenever a hooked runtime method is called, it loads the ReFrameworker configuration along with the contained rules ("items") and acts accordingly.

Managing the configuration file along with its rules is done via the ReFrameworker dashboard. Using the dashboard, you can define a set of rules that the Android runtime will obey. The dashboard will then generate a config file which the runtime will later parse and act accordingly.

For example, it starts with loading a config file which can be either loaded from local file or directly from the connected Android device. After clicking either of the "load config" buttons (figure 13), the dashboard will immediately mark all the loaded rules and allow the user to enable / disable them and also to configure them.

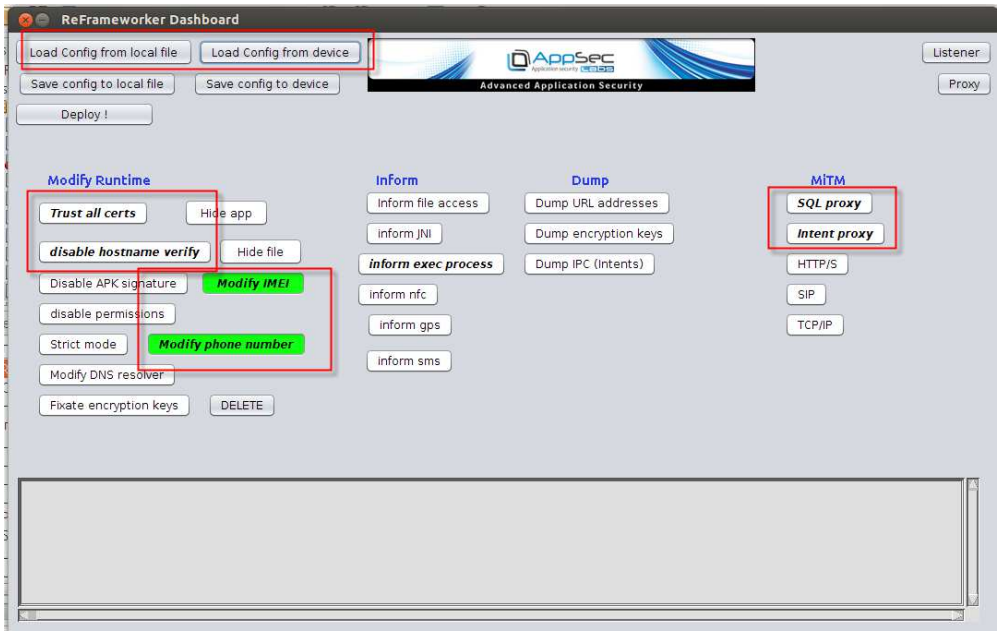


Figure 13 – loading rules from config file

After the file is loaded, the dashboard marks all the defined rules with **bold**, and highlights all rules which are also enabled as ***green***.

Then the user can choose which kind of behavior he wants from the runtime – for example, he can turn on sniffing of important information, bypass of certain logic, doing some string replacement, sending some data to the ReFrameworker dashboard and so on.

After that, the user can save the new configuration (figure 14). If the user chooses to save it into the device, from now on the device will behave according to that rule.

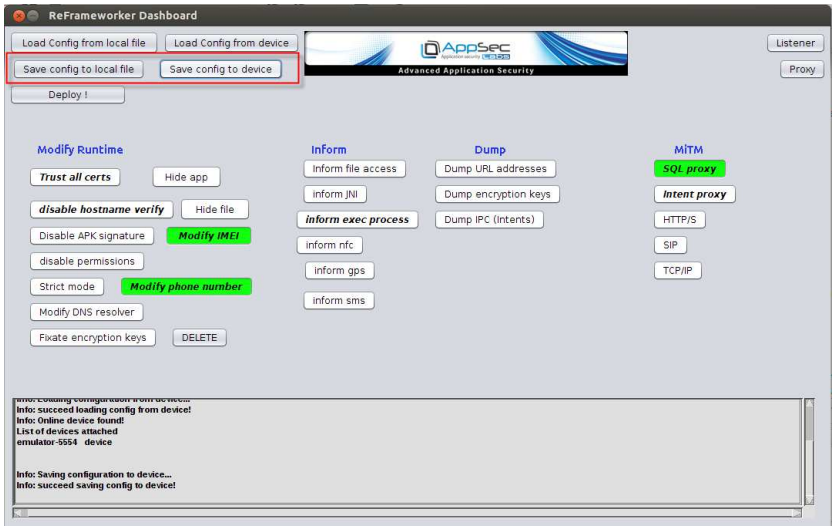


Figure 14 – saving rules to config file

Configuring the behavior of each rule can be achieved by clicking on the rule's item, and selecting "configure" from the sub menu as can be seen from figure 15.

Figure 15 – configuring an item

Then, a new window will appear, containing the values of that rule. Each rule has the following properties:

- Name – the name of the rule
- Enabled – is it enabled?
- Calling method – the name of the runtime method upon which this rule should apply
- Mode – can have 3 possible values – Send, Proxy, or Modify.
    - Send – send the hooked content to the ReFrameworker dashboard
    - Proxy - let the user control the value of the hooked content by using a proxy-like UI
    - Modify – replace a particular content with another content
- Value – specify the condition for the hooked content. An asterisk (*) means always.
- toValue - specify the action for the hooked content. An asterisk (*) means always.

Below you can find an example of how rules can be configured (figure 16):

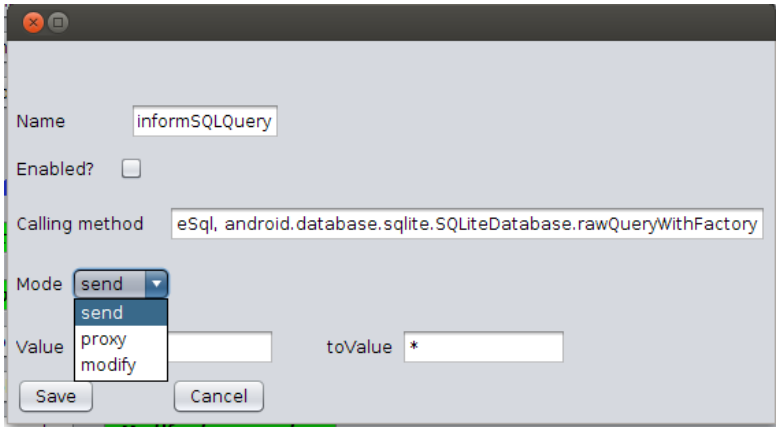


Figure 16 – an example for rule configuration

For example, here is how the config file will look like after generating new rules for the runtime - the following is an example of "Reframeworker.xml" configuration (figure 17):

```
<config>
<generalItem>
  <ip>10.0.2.2</ip>
  <port>6666</port>
</generalItem>

<item>
  <name>trustAllCerts</name>
  <enabled>true</enabled>
  <condition>true</condition>
</item>
<item>
  <name>setPhoneNumber</name>
  <enabled>true</enabled>
  <value>333333333333333333</value>
  <mode>proxy</mode>
  <caller>android.telephony.TelephonyManager.getLine1Number</caller>
</item>
<item>
  <name>informExecProcess</name>
  <enabled>true</enabled>
  <mode>send</mode>
  <caller>java.lang.Runtime.exec</caller>
</item>
<item>
  <name>informSQLQuery</name>
  <enabled>true</enabled>
  <mode>proxy</mode>
  <caller>android.database.sqlite.SQLiteDatabase.executeSql, android.database.sqlite.SQLiteDatabase.rawQueryWithFactory</caller>
</item>
</config>
```

Figure 17 – an example of config file content

The idea is to place this file at the specific place inside the runtime which our hooks will look for it. The location of this file should be at "/data/system/Reframeworker.xml" – so that our hooks which were pre-injected into the runtime will parse, load, and act upon dynamically while we can instrument them from the external of the Android device.

Since the device might communicate with the dashboard (sending some data, waiting for instructions, etc.), the dashboard contains a listener for incoming communication established from the device. Therefore, the dashboard contains a button for the listener (figure 18):

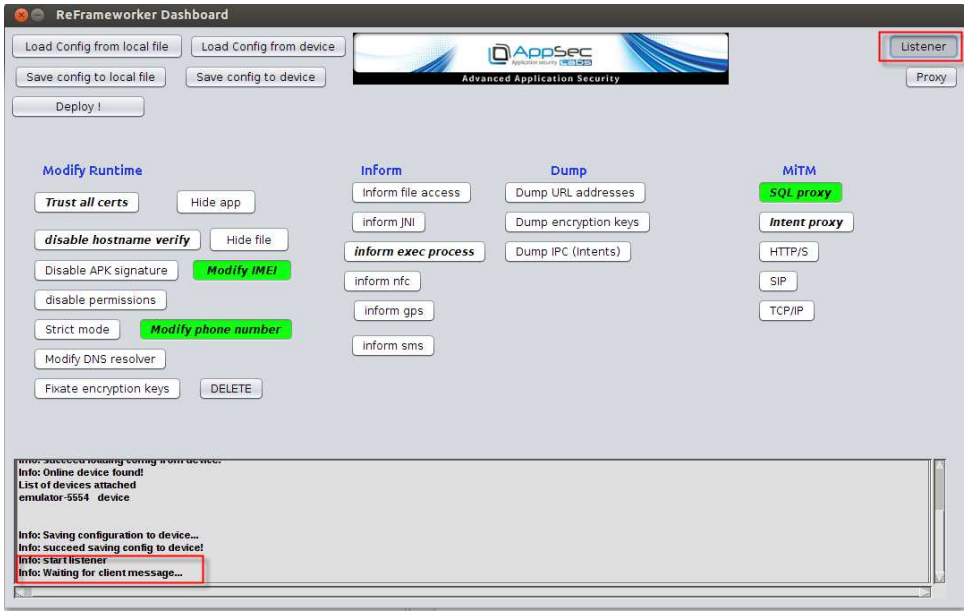


*Figure 18 – starting the listener*Now after starting the listener the dashboard is ready for any incoming messages.

AppUse also has an advanced feature that lets the user to intercept some internal information from inside of Android objects. We can do so by pushing a proxy between the Android application and the runtime. This is done in a very similar way to an HTTP proxy, but only that this time we're doing so at a very low level inside the Android runtime.

The hooks

The AppUse environment was compiled with lots of hooks at some key places. As part of the research, after finding out interesting places we want to control such as handling of files, communication, encryption, etc. we placed calls at those location to the ReFrameworker controller. The controller's responsibility is the check whether a rule is currently defined for this particular location, and if so it acts by its configuration.

For example, as part of the research of finding interesting locations to place a hook into we decided to have place a hook into the SQLiteDatabase at the "executeSql" method which all queries are passed through at. Hooking into this class will enable us to intercept all the local SQL queries sent from the application to its local DB. Our hook (which was placed inside the Android executeSql method inside the SQLiteDatabase class) will intercept this value and do whatever was instructed at the configuration.

Hooks are usually placed around an important value, such that if a rule is defined for this particular hook, then the controller's responsibility will be to do something with it. The controller can either **do nothing** and leave that value as is (in case no rule is defined or the rule is disabled), it can **send that data** to a remote location, it can allow the user to **break and modify that value at real time** (i.e. in a similar manner as a proxy) or it can do an **automatic replace** for another value.

For example, this is how the pre-loaded hook will look like when hooking at the executeSql method into the "sql" string parameter - the actual query that will be executed by the runtime, as requested from the upper level application (figure 19)

```
private int executeSql(String sql, Object[] bindArgs) throws SQLException {
//added
sql = controller.operateString(sql);

    acquireReference();
    try {
        if (DatabaseUtils.getSqlStatementType(sql) == DatabaseUtils.STATEMENT_ATTACH) {
            boolean disableWal = false;
            synchronized (mLock) {
                if (!mHasAttachedDbsLocked) {
                    mHasAttachedDbsLocked = true;
                    disableWal = true;
                }
            }
            if (disableWal) {
                disableWriteAheadLogging();
            }
        }

        SQLiteStatement statement = new SQLiteStatement(this, sql, bindArgs);
        try {
            return statement.executeUpdateDelete();
        } finally {
            statement.close();
        }
    } finally {
        releaseReference();
    }
}
```

Figure 19 – ReFrameworker hook that was pre-injected into the runtime

Suppose the relevant configuration rule for this was defined as "proxy" - Now each time this method is called, the device will send this data (the original query) to the proxy and will replace the original value with modified received value.

All it takes on the dashboard side is to operate the proxy (figure 20)

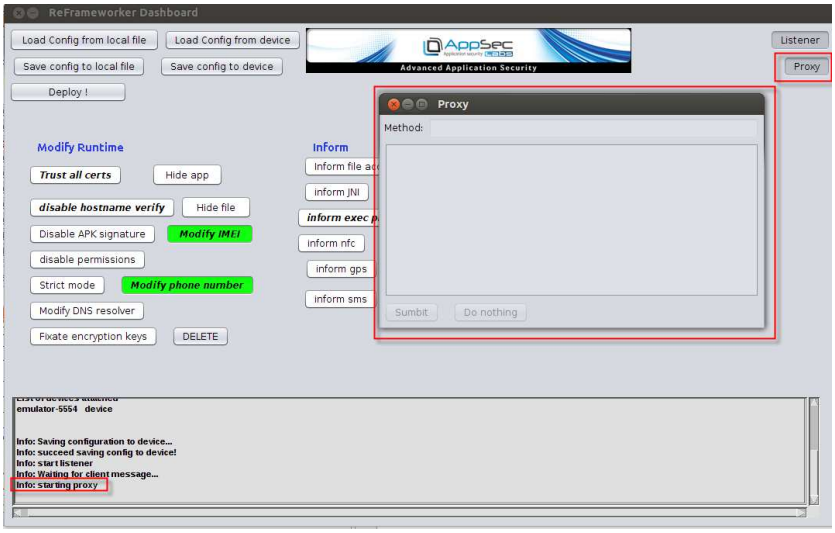


Figure 20 – starting the proxy

Now, when a message will be received, the proxy will wake up and give the user the opportunity to observe the message AND modify it – while the android app is waiting for the response! (Figure 21)

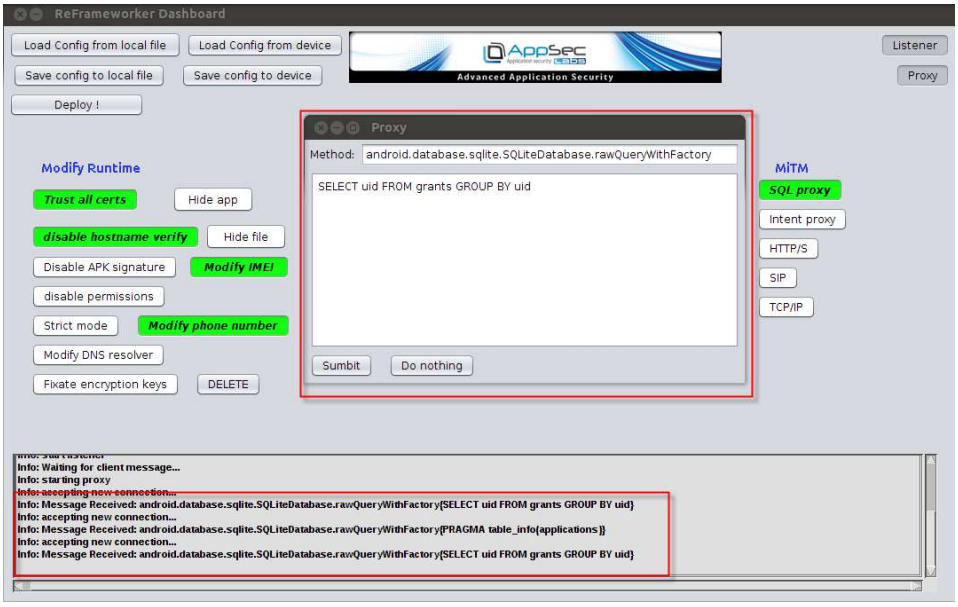
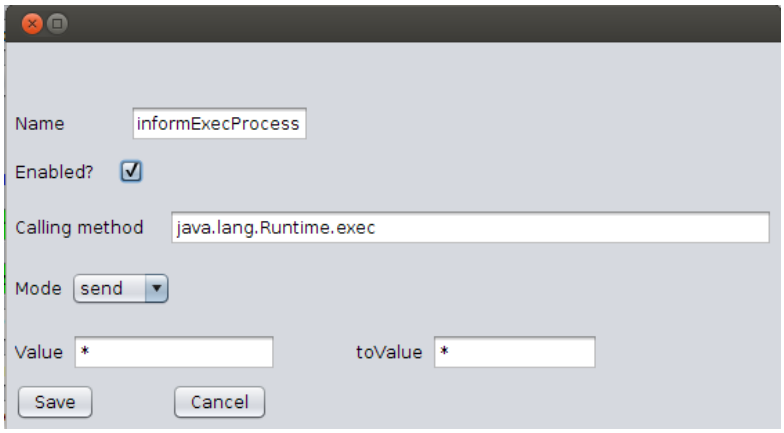


Figure 21- handling incoming message using the proxy

Configuration examples

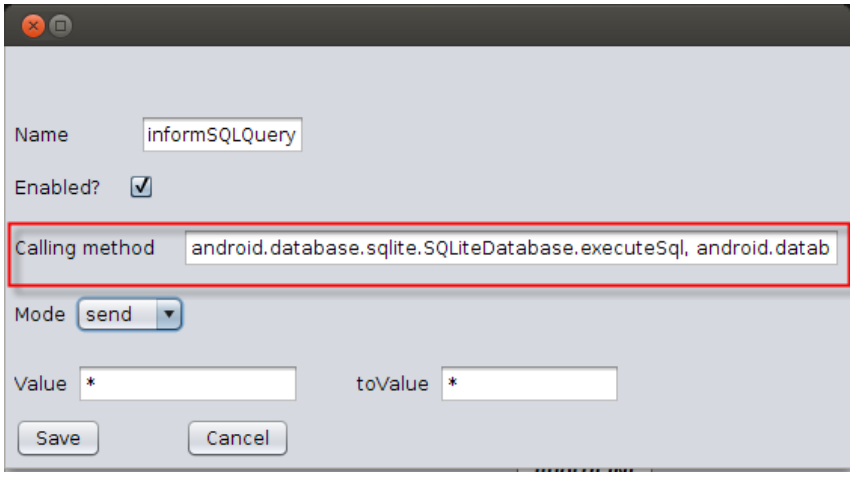
Let's take a couple of examples for common scenarios, and how the configuration should be set to achieve the required behavior. For each of the following examples, we'll demonstrate how its configuration should look like (as captured from the default rules that come with the AppUse Reframeworker config file), along with a brief explanation for its settings.

Example #1 – send the value of all executed commands to the dashboard



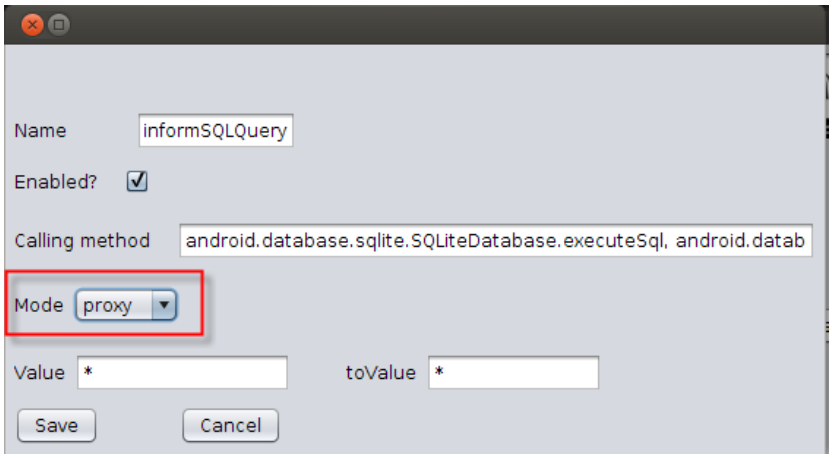
Explanation – mode is set to "send" since we want to send this data. Value is *, since we want to send all commands. toValue is not used in this context but is set to *just in case. Calling method is set for the relevant hooked method.

Example #2 – send the value all executed SQL queries to the dashboard



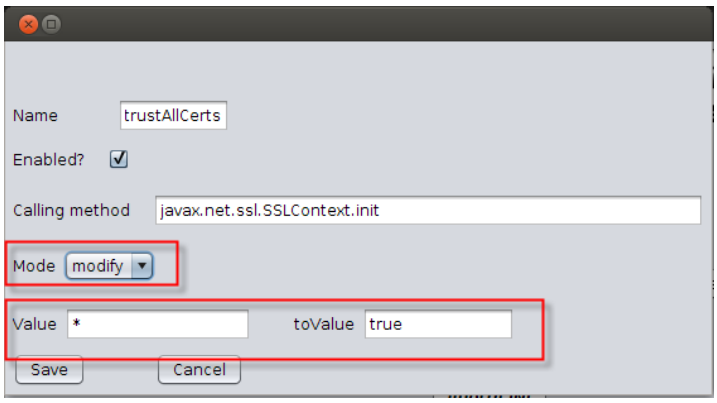
Explanation – calling method was set to the specific methods responsible for SQL queries. Other values stayed the same (compared to previous example).

Example #3 – proxy (break and modify) the value all executed SQL queries to the dashboard



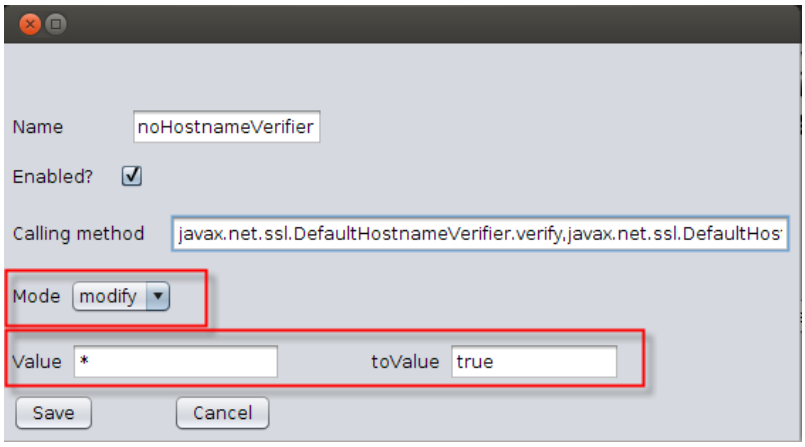
Explanation – mode is set to "proxy" since we want to modify this data at realtime. Other values stayed the same (compared to previous example).

Example #4 – trust all certificates



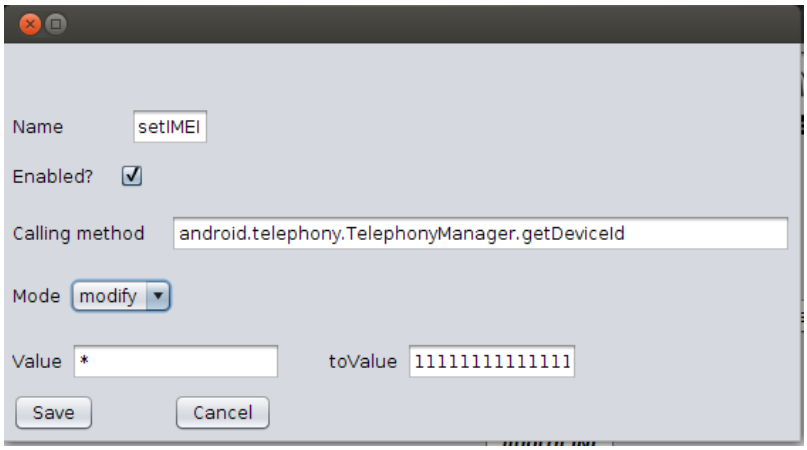
Explanation – mode is set to "modify" since we want to replace the hooked value (specifically, the Boolean value of whether the certificate should be trusted). Value is *, since we want to replace all possible values (whether the cert is ok or not). toValue is set to true since we want to always trust the certificate. Calling method is set for the relevant hooked method.

Example #5 – disable hostname verification



Explanation – quite similar to the previous example. The only difference is the value of calling method which is the hooked method responsible for hostname verification.

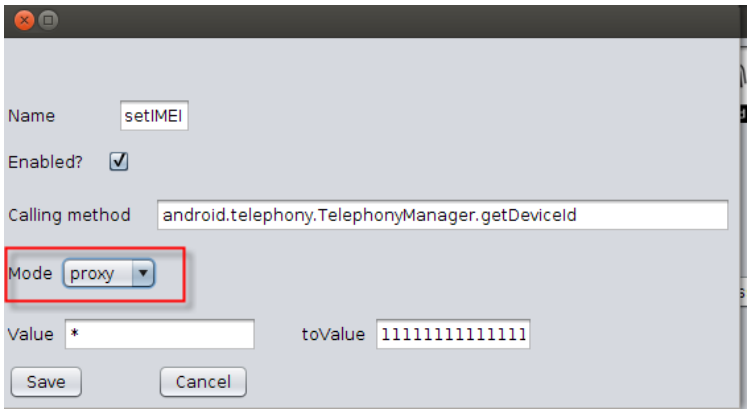
Example #6 – replace the value of the phone IMEI number with another value



Explanation – mode is set to "modify" since we want to replace this data. Value is *, since we want to replace all possible values. toValue is set to "111111111111" which is the value we want to set in this example. Calling method is set for the relevant hooked method.

Note – if wanted to replace only a specific number, all we needed to do is to set it as "value" (rather than using * in this example).

Example #7 – proxy (break and modify) the value phone IMEI number



Explanation – mode is set to "proxy" since we want to modify this data at realtime. Other values stayed the same (compared to previous example).

Of course, this is just a very brief introduction to all the ReFrameworker strength, as there are lots of other rules AppUse can manage and for each one of them there are lots of different settings to play with.

Network Analysis with AppUse

A big part of pentesting Android applications is done by inspecting the transportation from the app and tampering it. In order to achieve the functionality, AppUse has preinstalled and configured tools to be used in the penetration testing.

Proxied Emulator

To ease the work in monitoring and analyzing network traffic, AppUse comes with an option to fire up the emulator proxy-initialized with Burp proxy with a click of a button or as standalone (figure 22).

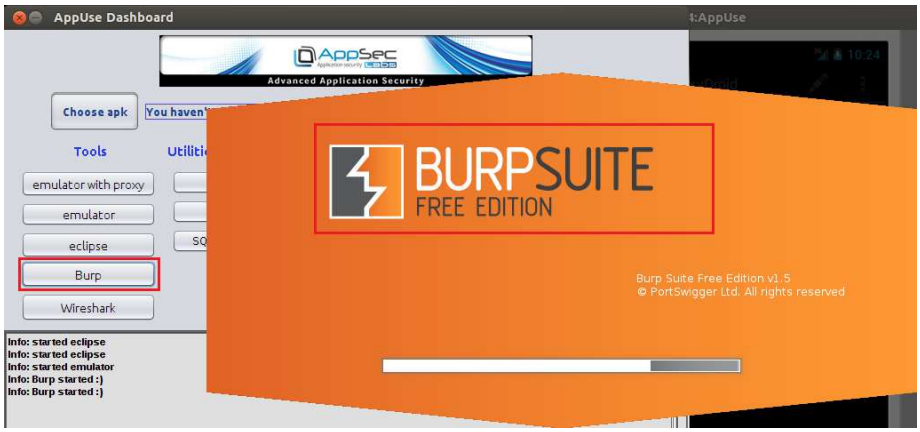


figure 22 - Launching Burp from the dashboard.

Burp Suite is a leading HTTP proxy aimed and designed for penetration testers. AppUse makes use of Burp to view and tamper HTTP traffic and facilitate various attacks. Burp also provides many tools to analyze data and HTTP traffic. It is extendable via many plugins to suit most of the web-based technologies.

In order to analyze encrypted network traffic (HTTPS), the emulator's Key Store comes with Burp's Root CA Certificate pre-installed so it will be trusted on the emulator's OS and enable testing HTTPS without having applicative errors involved (figure 23).

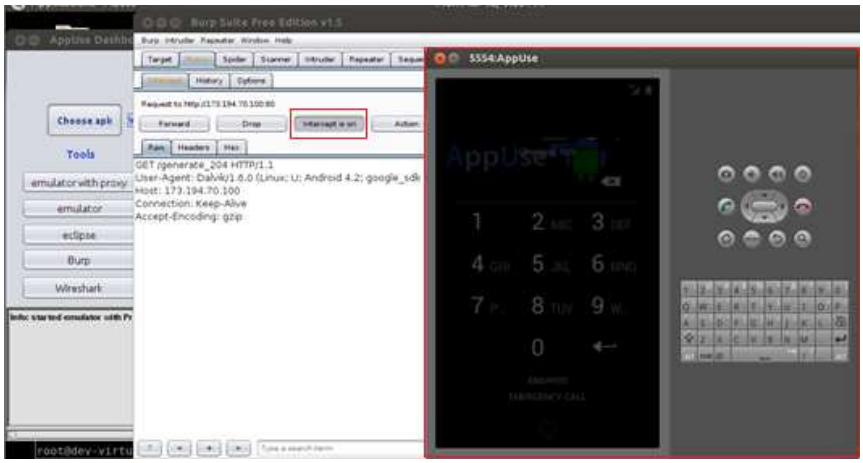


figure 23 - Burp is automatically linked to intercept communication from the emulator on 127.0.0.1:8080

Wireshark

AppUse comes with Wireshark sniffer (figure 24) preinstalled and launchable from the dashboard. Wireshark is the world's foremost network protocol analyzer. It lets you capture and interactively browse the traffic running on a computer network from all protocols and network layers. Wireshark enable the pentester to deeply inspect all the traffic on the device without the limitation of HTTP-based protocols.

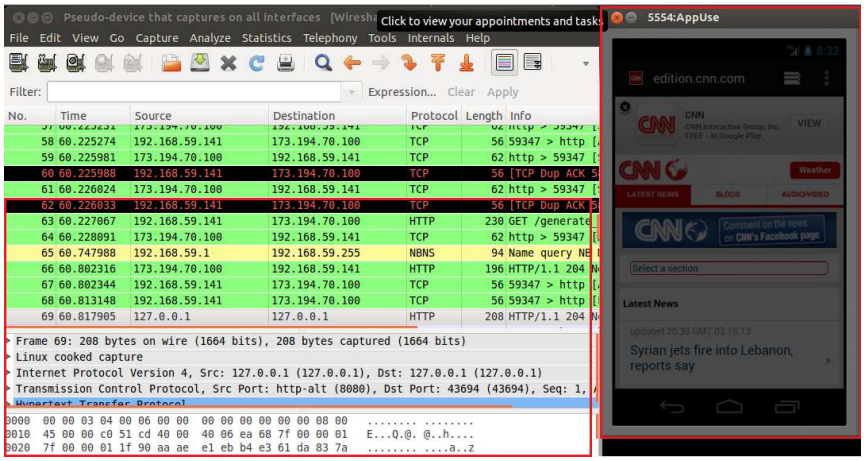


figure 24 - In action: wireshark sniffing the Browser app surfing to CNN.com

Decoding and Reverse Engineering APK's

AppUse includes the most advanced tools used to decode and reverse engineer APK's. Once an APK is loaded to the dashboard all the tools are preconfigured to use it and all the tools and frameworks are leveraging the pentester to reach full coverage.

Extracting APK

APK's are encoded zip archives. Upon opening an APK, there is a predetermined files and directories structure that let the pentester to learn what is hidden under the hood. Among other, a pentester can learn about open broadcast receivers, the code being the application, the resources it uses, the permissions it asks for and more…

The first step in the analysis is to extract the APK (figure 25). The Dashboard allows it to be done in one click.

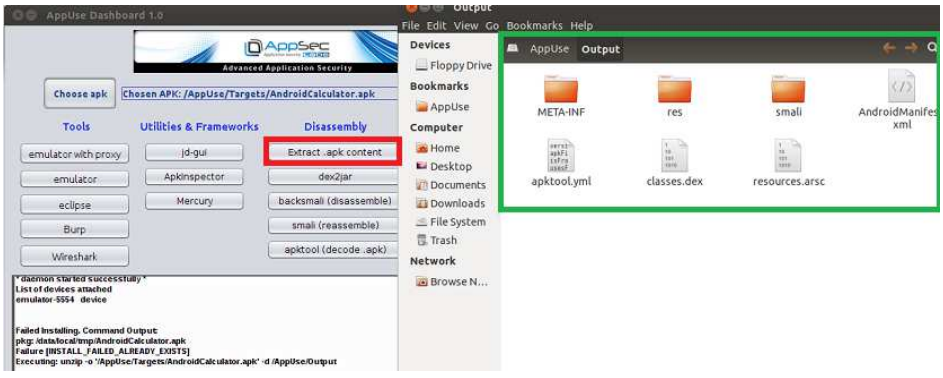


Figure 25 - Extracting APK – result in APK contents being presented on the Output directory.

Converting Dalvik to Java code – dex2jar

The classes.dex file in the application's APK is Java's .class equivalent which contains all the classes' hierarchy and Java code compiled into Dalvik byte code. dex2jar is a tool that converts the Android's classes.dex file to .jar file, which lets the pentester to disassemble it via Java disassembling tools, such as jd-gui.

By clicking on dex2jar (figure 26), the Dashboard will automatically seek an extracted APK on the Output folder and choose its classes.dex file. dex2jar will save in the Output directory an equivalent .jar file to be used further in the testing.

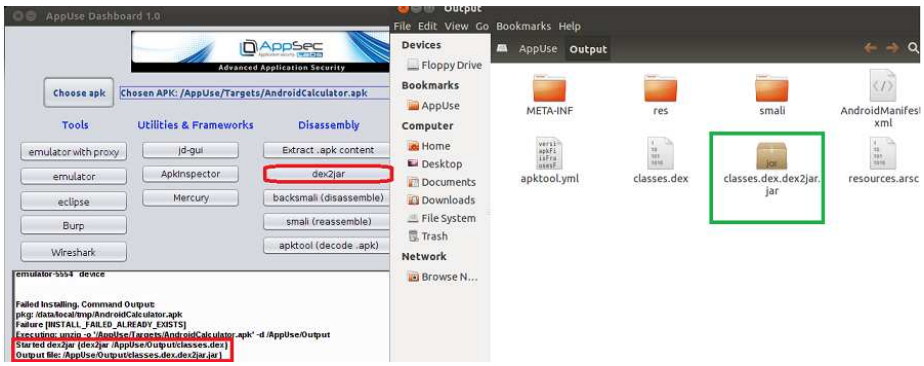


figure 26 - dex2jar running on an example APK – AndroidCalculator. The result is Java-equivalent jar file.

Disassembling Java Code with JD-GUI

JD-GUI is a framework aimed to disassemble .jar files. Once a .dex file had been converted to .jar, JD-GUI framework is ready to disassemble the code. The pentester, by using JD-GUI, will be able to audit the application code to find hidden secrets and logic (figure 27).

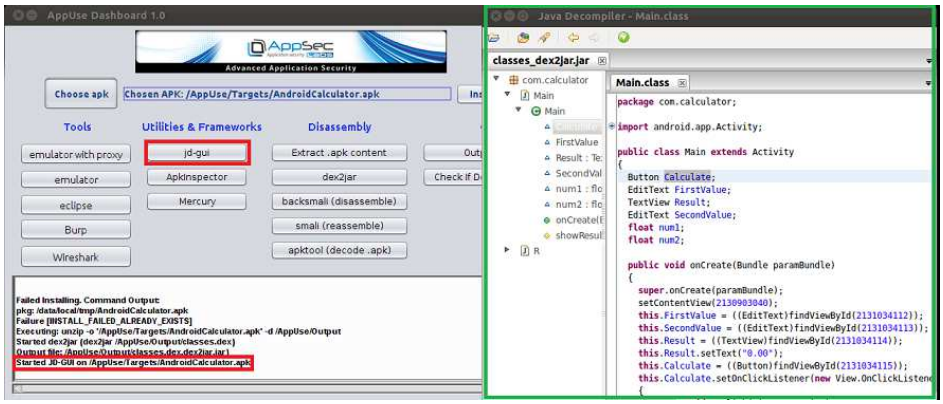


Figure 27 - Disassembling Java code of chosen Android application via JD-GUI.

Decoding the APK via APKTool

Extracting APK does not take care of Android's encodings. For instance, when opening an APK, the AndroidManifest.xml file will be encoded so it won't be in a human-readable format. In order to unveil the encoded secrets, APKTool can be used.

APKTool (figure 28) is able to decode APK files to a human-readable format. For instance, the following shows an attempt to open AndroidManifest.xml file before using APKTool.

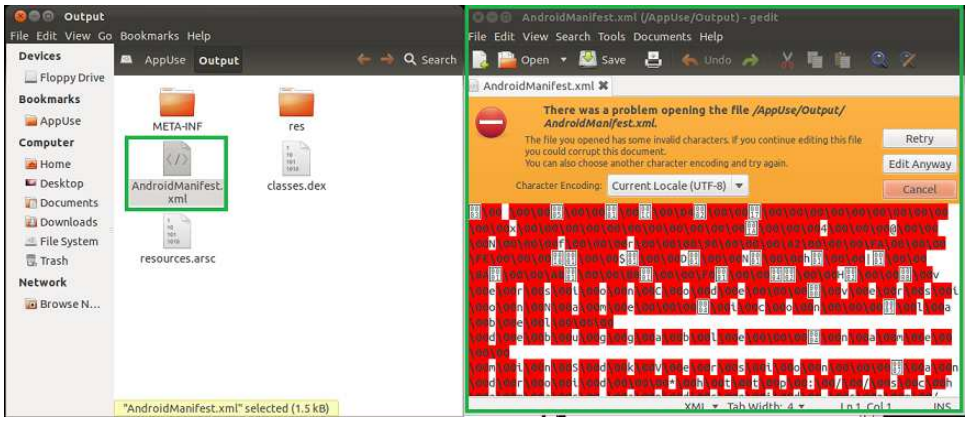


Figure 28 - AndroidManifest.xml before using APKTool

Using APKTool decodes the APK successfully, and gets the AndroidManifest.xml to a human-readable format (figure 29):

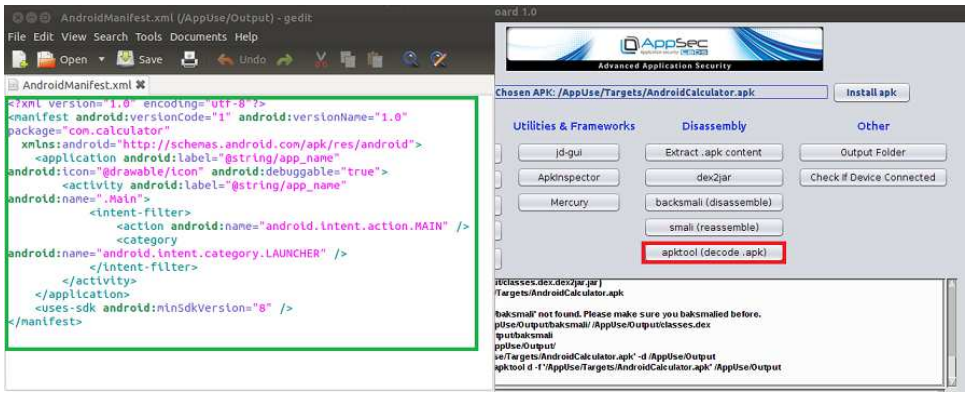


Figure 29 - The AndroidManifest.xml after decoding the APK.

Modifying APK's

In some security researches it is necessary to patch the code base of an application. While with existing source code it can be done easily, without existing source code the task becomes harder. AppUse contains all the tools necessary to disassemble and patch Android applications code.

There are many applications to modifying the code base of an application. The ability will let the pentester to hook functions, inject code of his own or change the resources being used by the application.

The following section will show how the pentester can disassemble and modify an application via AppUSe.

Disassembling via baksmali

baksmali is a tool used to disassemble an APK's Dalvik byte code. Via baksmali, a researcher can view the Dalvik assembly of the application and modify it with a human-readable format.

AppUse Dashboard has a baksmali button that with one click will disassemble the APK and will output it to /Output/baksmali folder (figure 30)

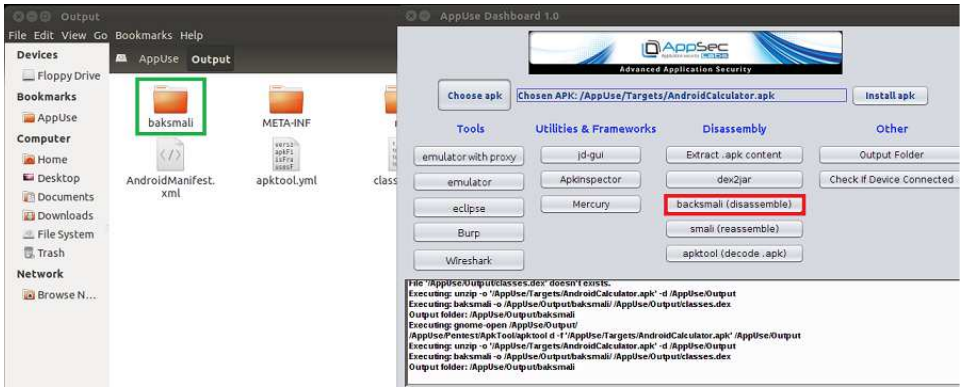


Figure 30 - Clicking the baksmali button – will create the /Output/baksmali folder with all the Dalvik assembly code of an application.

Once baksmali had been used on an APK, the /Output/baksmali folder will contain all of its Dalvik assembly code in a human-readable format. The assembly can then be modified and have added instructions of the pentester (figure 31).

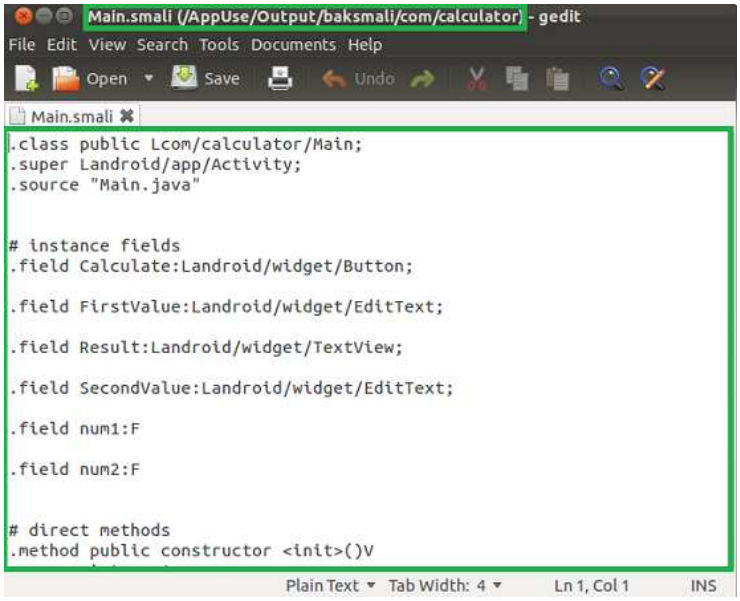


Figure 31 - baksmali output – human-readable assembly of the application. Editable by any text editor.

Reassembling with smali

baksmali gives the researcher the power to have human-readable dalvik assembly code and have the chance to edit it with any text editor he wishes. Smali is a tool to complete the puzzle to reassemble the code again.

With smali, the researcher can perform changes in the application's assembly code and recompile it to a new .dex file. Once the new .dex file will be applied to an APK, the changed code will be patched and once the APK will be installed the changes in the code will be applied in runtime. Using this feature can leverage security researches to a whole new level.

AppUse Dashboard have the smali button which will take an existing code base created by baksmali, with all modified changes from the /Output/baksmali folder and will recompile it altogether to a new .dex file with one click of a button (figure 32). The new dex file will be located at /Output/smali_dex.dex.

Figure 32 - Smali function – reassembling all smali code under /Output/baksmali folder to a new .dex file at /Output/smali_dex.dex.

Signing APK

APK needs to be signed in order to run on a device. Since modifying an APK will make the original APK signature no longer valid, the researcher needs to sign is APK. This can be done by using the SignAPK tool located at /AppUse/Pentest/SignAPK. The sign.sh script needs to parameters: [existing_apk] and [signed_apk], where [signed_apk] indicates where to create the newly signed APK file (figure 33).

AppUse had preconfigured encryption keys for the sake of simplicity. The signapk.jar file contains more options if the researcher wants to use his own encryption keys. More details about the tool can be found at: http://code.google.com/p/signapk/

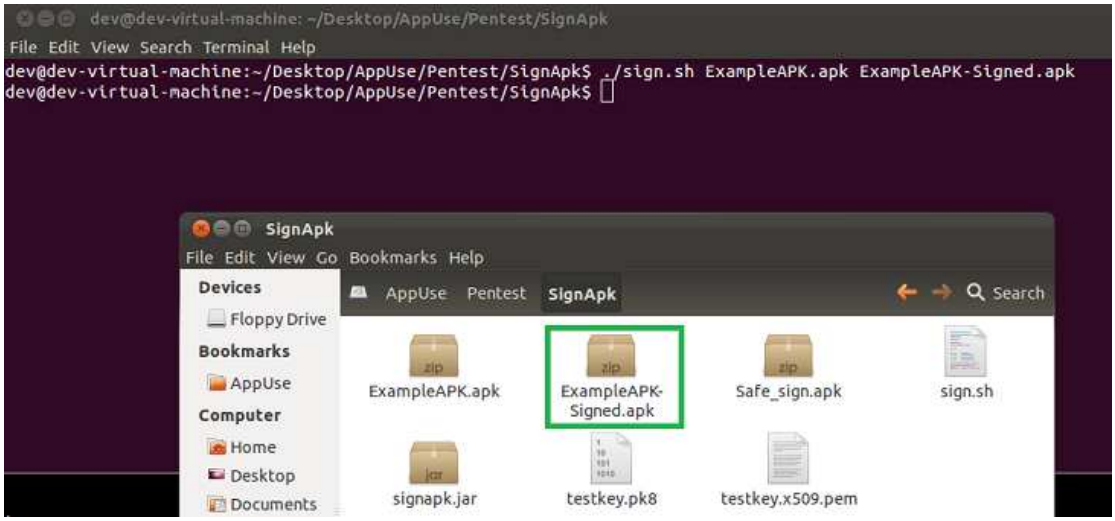


Figure 33 - SignApk – Shows how to sign an APK. The ExampleAPK.apk is the unsigned one and the ExampleSPK-Signed.apk is a newly created and signed APK by the tool

Going through the runtime

AppUse has preinstalled all the tools needed to go through the application runtime. The following section will guide the basics on how to look up an application runtime via AppUse.

Using ADB anywhere

ADB is the most important tool to observe an application runtime on a live environment. It lets the pentester look through the application's internal storage, broadcast messages and more. AppUse acknowledge the significance of ADB, thus it is embedded in the PATH environment variable and accessible from anywhere on the terminal.

The following figures will show some of the basic operations you can do with ADB.

Figure 34 - Opening a unix shell on the emulator

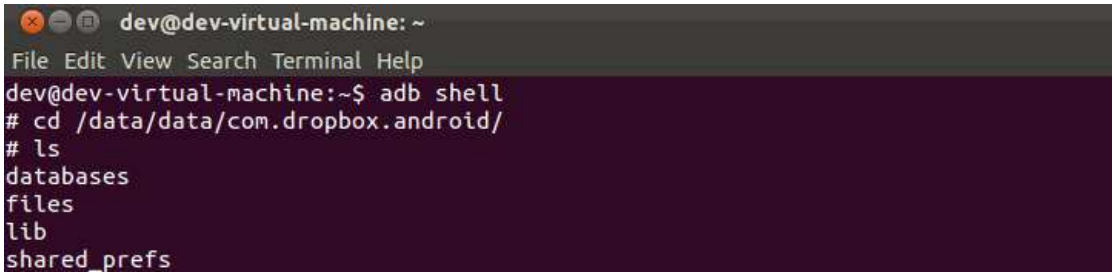


Figure 35 - Looking at an application internal storage

Figure 36 - Taking a database from the application's internal storage to the local machine

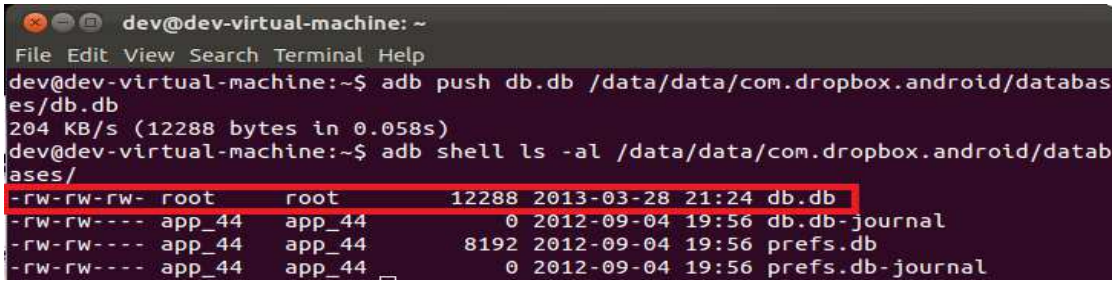


Figure 37 - Overwriting a file on the application's internal storage with a file from the local machine

Figure 38 - Broadcasting intents

Analyzing Internal Databases

Android allows an application to use an internal SQLite database for its private storage. Inspecting a database is always part of a penetration test, thus AppUse has SQLite browser bundled. The following shows how an application database can be extracted from the runtime local storage and inspected via the SQLite browser.

Pulling the database from the internal storage via adb pull (figure 39):

Figure 39 – pulling the database file

Browsing the database via SQLite browser (figure 40):

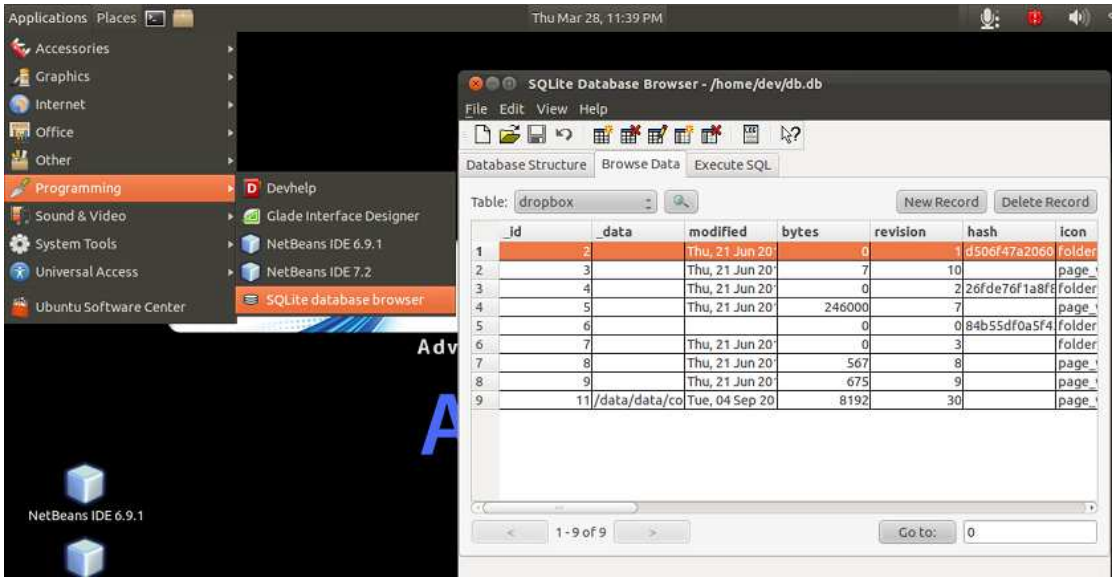


Figure 40- using the SQLite browser

The /Pentest Folder

AppUse is a project that is found under constant development. One of the main goals is to always be up to date with the latest attack tools, enabling a researcher to achieve full attack coverage of a given application.

The directory structure of AppUse contains the /Pentest folder, which is where all the tools are allocated. A brief overview of the folder will give us this (figure 41):

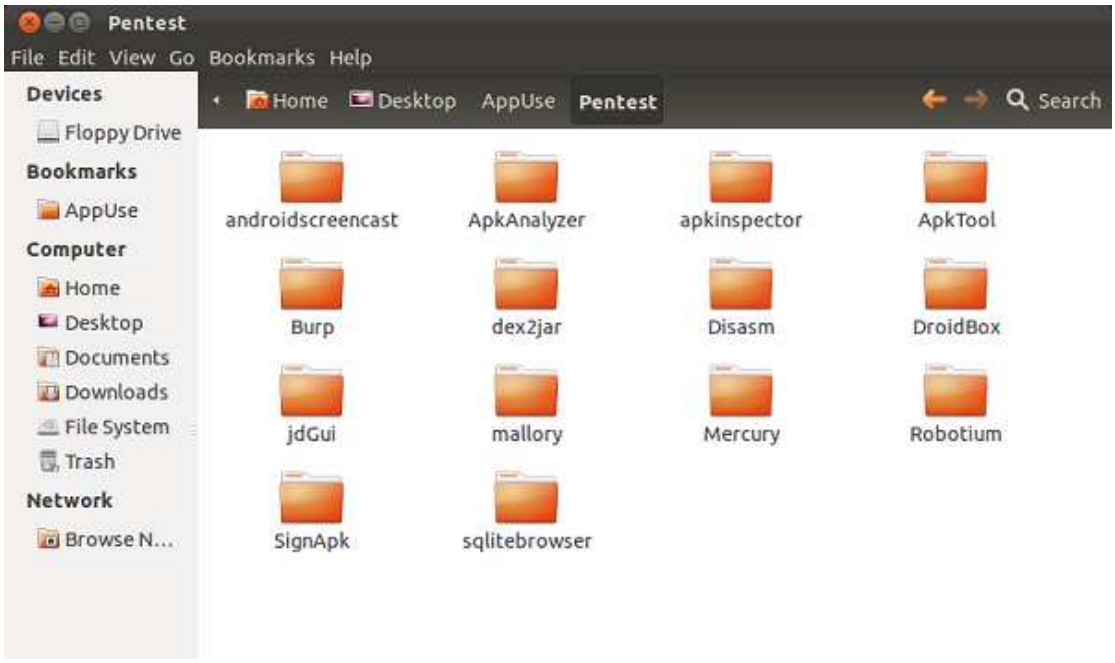


Figure 41 – the /Pentest folder

As you may have noticed, not all the tools are currently present in the Dashboard. While the Dashboard has many functionalities that in a standard research will fully cover all the researcher needs, some others are still in development and not embedded yet to the Dashboard.

AppUse won't stop you from using those, as we are aware that for some researchers it is a need. Those weapons will be found under the /Pentest folder.

For instance, the following shows the APK Analyser in action (figure 42):

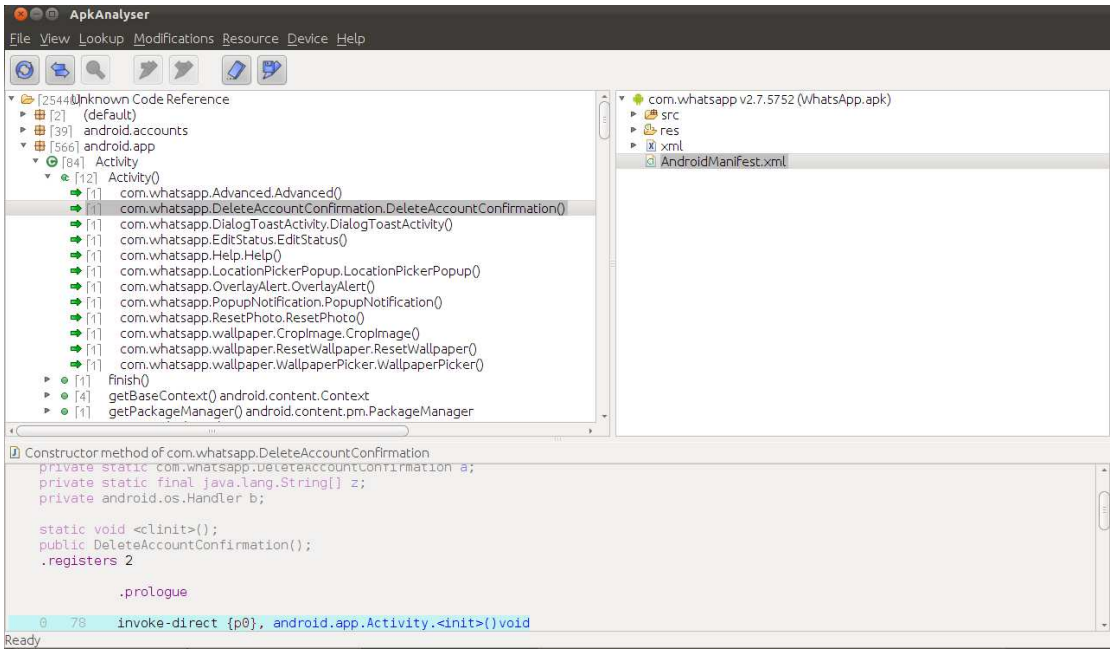


Figure 42 – the ApkAnalyzer tool

Of course more tools are available, such as Robotium, DroidBox and SQLite Browser. This folder is constantly updated and new weapons are added all the time.

Link to AppUse

AppUse can be downloaded here: https://appsec-labs.com/AppUse