



Mobile Pentest Environment

AppUse is designed to be a weaponized environment for Android and iOS application penetration testing. It is a unique and rich platform aimed for mobile application security testing.

<https://www.appsec-labs.com>

info@AppSec-Labs.com

Last updated September 2017

Information in this document is subject to change without notice. Companies, names, and data used in examples herein are fictitious unless otherwise noted. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of AppSec-Labs.

Contents

AppUse - Overview	3
AppUse - OS	3
Credentials	3
Terminal and Command Line Tools.....	3
Development Tools	5
AppUse Directory Structure.....	5
AppUse - Dashboard.....	9
Overview.....	9
Android Dashboard Structure:	11
General.....	11
Home	14
Android Device	14
Tools	18
Reversing	20
Application Data	23
ReFrameworker	24
Vulnerable Apps.....	24
AppUse - Runtime Modifications and Inspection via AppSec ReFrameworker	25
How it works – an overview	25
The hooks	28
Configuration examples.....	30
iOS Dashboard Structure (Beta Version):.....	34
General.....	34
Home	35
Analyze Applications.....	35
Console/NSlog	36
Runtime Hooking	36
Jailbreak.....	36
Tools	36
iNalyzer agent CLI.....	37

AppUse - Overview

AppUse is designed to be a weaponized environment for Android and iOS application penetration testing. It is an OS for mobile application pentesters that contains tools and hooks for easy application control, observation, and manipulation.

AppUse can be downloaded from here:

<https://appsec-labs.com/appuse>

AppUse - OS

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

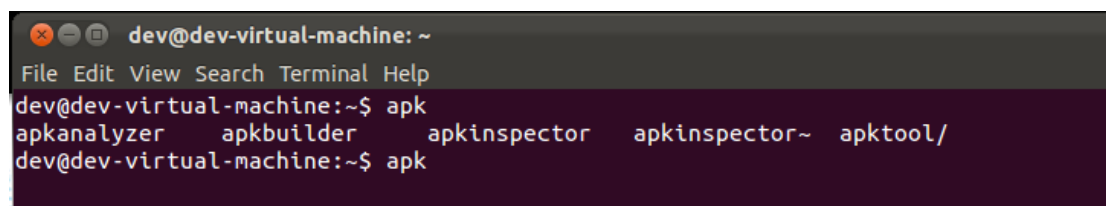
Credentials

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

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

Terminal and Command Line Tools

The environment variables have been equipped with tools for the Android and iOS 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).



```
dev@dev-virtual-machine: ~  
File Edit View Search Terminal Help  
dev@dev-virtual-machine:~$ apk  
apkanalyzer  apkbuilder  apkinspector  apkinspector~  apktool/  
dev@dev-virtual-machine:~$ apk
```

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

Aliases

A few more aliases have been added in AppUse > 1.8, to the terminal in order to help the pentester work faster.

Signapk <apk name>

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 the APK. This can be done by using the SignAPK tool located at /appuse/pentest/SignAPK/sign.sh. The script needs one

parameter: [existing_apk], thus, the script will create a signed APK named <apkname>_signed.apk in the current folder and delete the old unsigned APK.

AppUse has 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/>

The alias “signapk <apk path>” is a shorter way to execute the /appuse/pentest/SignApk/sign.sh in order to sign the modified apk file.

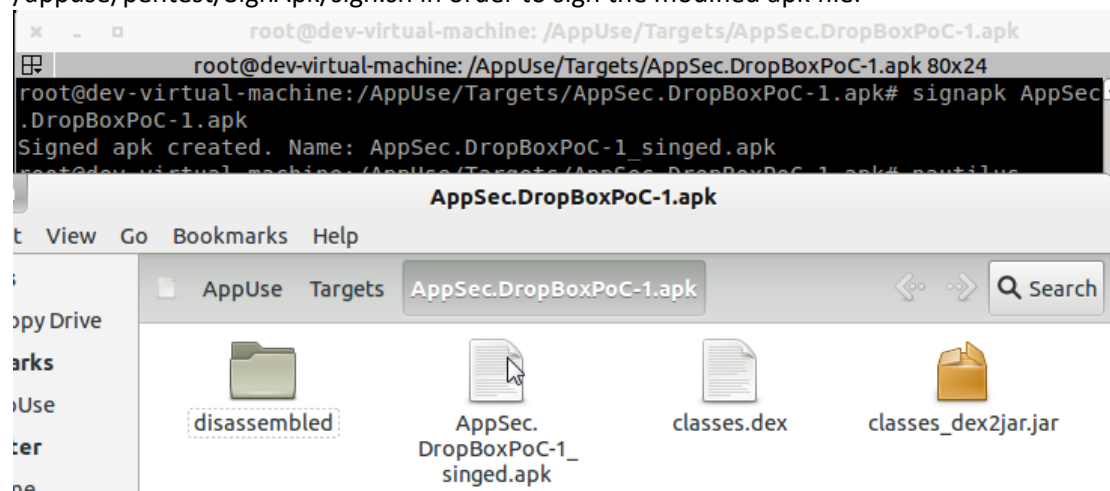


Figure 4: the SignApk – Shows how to sign an APK. The ExampleAPK.apk is the unsigned one and the ExampleSPK-Signed.apk is a newly created and signed APK by the tool.

Search <string>

The search alias is a Python script using a bash to search strings in files, it is very helpful when a pentester is searching for a method name or any string in files(e.g. smali code) (figure 5).

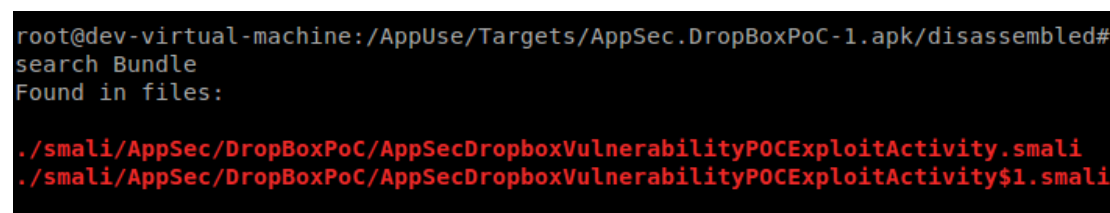


Figure 5: the search alias, the “Bundle” string found in 2 files.

Rootdevice

The rootdevice alias is a quicker way to execute the /appuse/emulator/rooting/root.sh in order to root the device.

Jdgui <jar file>

The jdgui alias is a quicker way to execute JD-GUI.

sublime <file>

The sublime alias is a quicker way to execute sublime text editor.

Development Tools

The AppUse environment includes Android development and debugging tools that can come in handy while performing applicative penetration testing. The environment has a preinstalled version of Eclipse and ADT (figure 6) 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 in special scenarios.

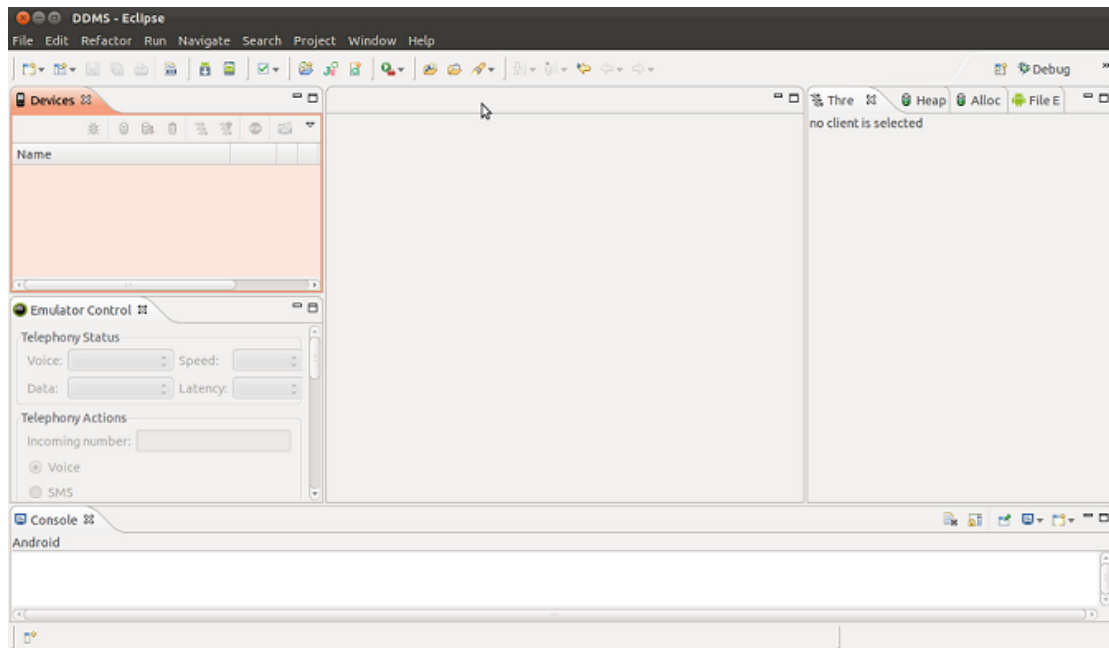


Figure 6: Eclipse with ADT. Useful in writing cross-application vulnerabilities exploits.

AppUse Directory Structure

AppUse Directory is located in the path “/appuse” and also appears as a shortcut on Ubuntu desktop. AppUse directory includes the following directories:

- .android – Includes the Android directory that includes 2 directories, a directory containing the SDK and the ReFrameworker directory which contains the entire ReFrameworker platform.

- .emulator – Includes the Emulator directories, such as sdcard, wallpaper, rooting etc.

- logs – Includes the AppUse dashboard log files.

- pentest – Includes multiple directories of pentesting tools , such as Burp, apktool, jdgui, mercury, dex2jar, etc.

- targets – The most important directory in the AppUse folder. This includes the targets apks / application data directories described in the next paragraph.

- ios – The iOS target’s folder

The Targets directory:

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 Targets folder. The Targets directory (will be elaborated later on this document) includes all the target's application folders which contain the output from all the tools in the dashboard.

While using one of the features in the Reversing section or Application Data section, the targeted APK/App's data will show its actions output in the target app folder inside the Targets folder. For example, if you will use the disassemble feature of APK named "AppSec.DropBoxPoc-1.apk" the output will be saved in the path /appuse/targets/AppSec.DropBoxPoc-1 /disassembled/ (figure 7).

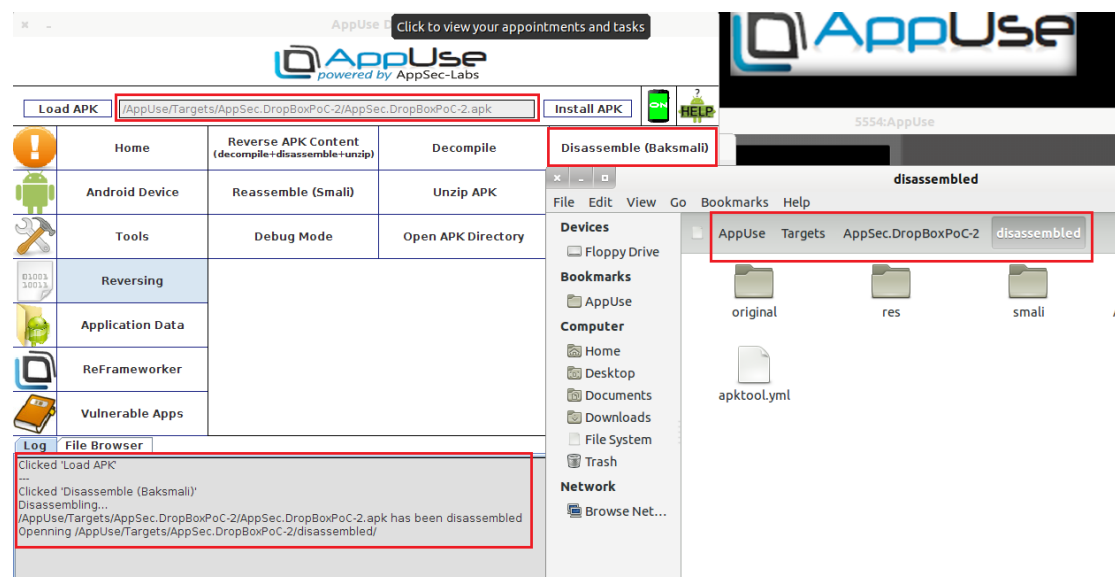


Figure 7: Targets folder – the Dashboard's workbench.

The Pentest directory

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 view over the folder will give us this (figure 8):

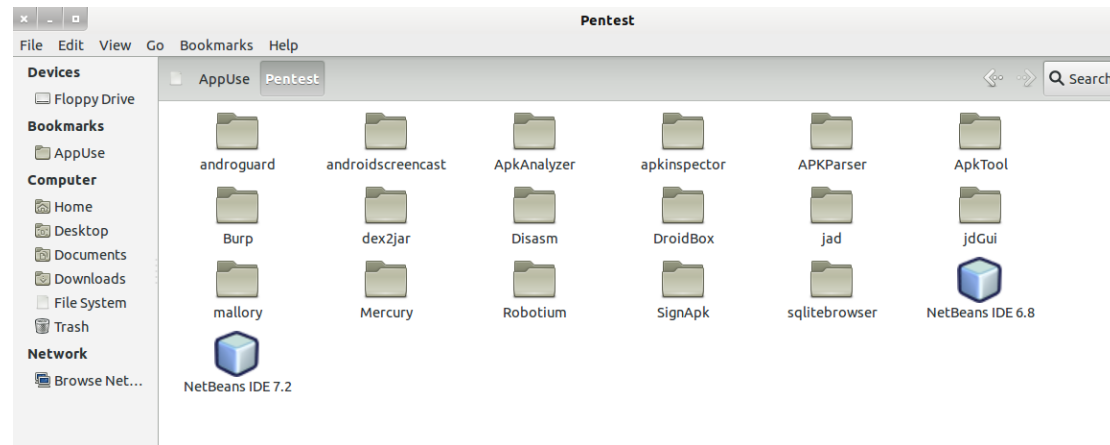


Figure 8: The /pentest folder.

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

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

For instance, the following shows the APK Analyzer in action (figure 9):

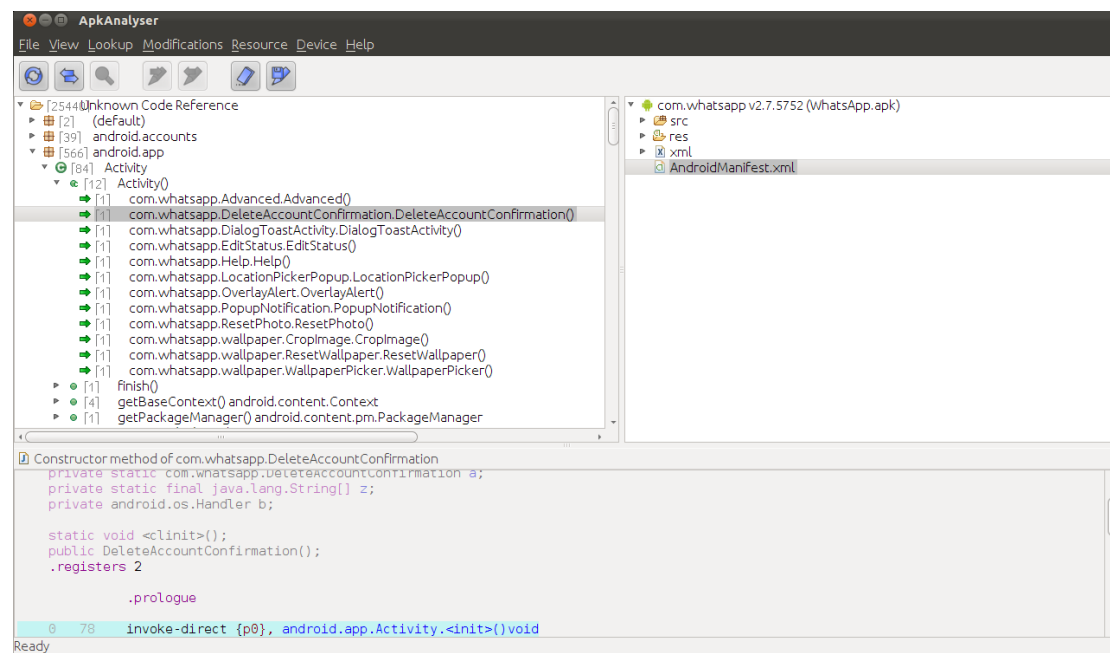


Figure 9: The ApkAnalyser tool.

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.

AppUse - Dashboard

Overview

The dashboard is the heart of the AppUse testing environment. The new dashboard now offers the option to test both Android and iOS applications via a GUI which organizes the testing tools and runtime environment that will be used during the research.



Figure 10: The new Appuse pro dashboard

The dashboard will put the puzzle together by linking all the data from all different tools together and will save precious time in its special functionalities that will concatenate several actions together, and 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 11):

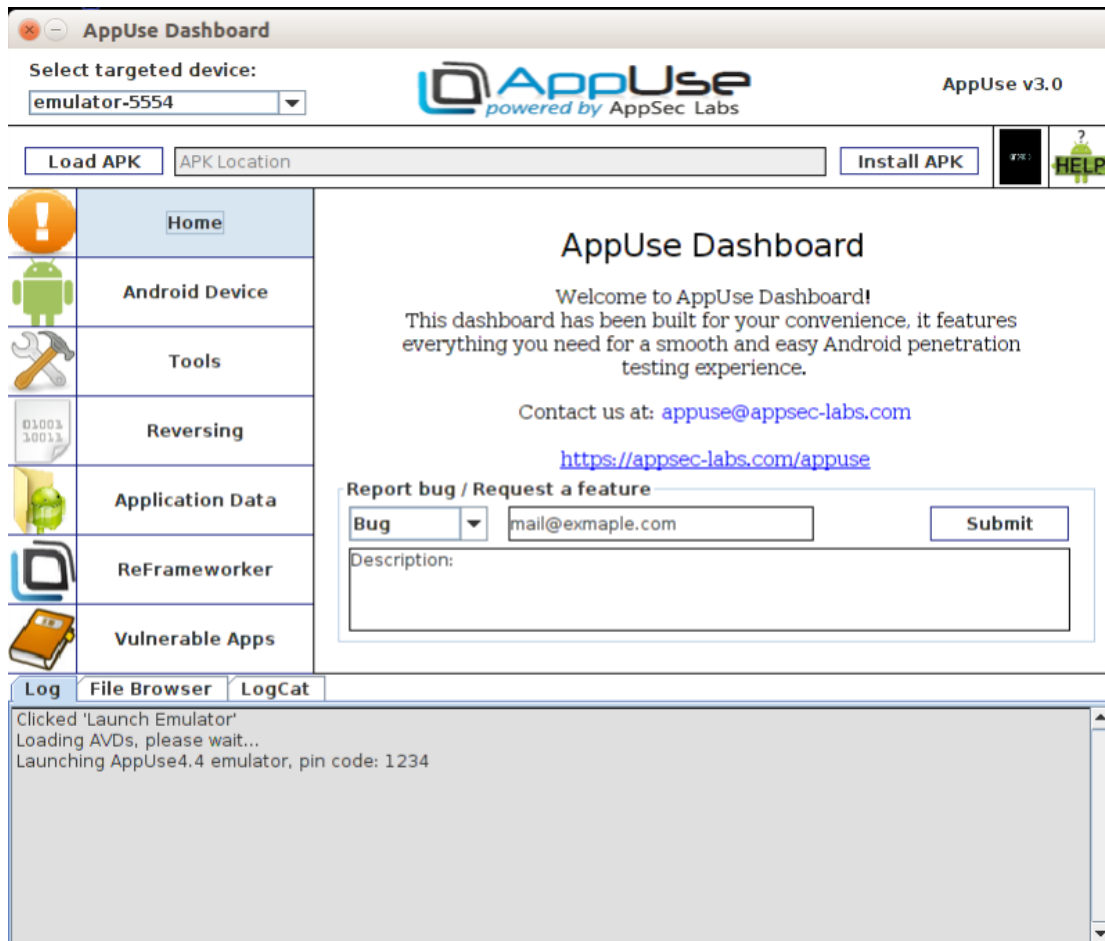


Figure 11: Launching the dashboard.

Android Dashboard Structure:

The dashboard is divided into 7 sections; each one is aimed to a specific purpose.

General

In the dashboard, the APK is the king. The AppUse dashboard has a goal to have researchers be able to start working with one-click actions. In order to achieve this goal, the dashboard is designed to operate on an APK and will use it while invoking its other actions.

Select targeted device

From AppUse version < 3.0 a new feature was implemented in order to support real devices and multiple Android devices (emulators or real devices), it is possible to connect real devices and turns multiple Android emulators and easily select the targeted device that will be attached and targeted via AppUse Dashboard.

AppUse Dashboard

Select targeted device:
emulator-5554

AppUse powered by AppSec Labs

AppUse v3.0

Load APK APK Location Install APK

Home

Android Device

Tools

Reversing

Application Data

ReFrameworker

Vulnerable Apps

AppUse Dashboard

Welcome to AppUse Dashboard!

This dashboard has been built for your convenience, it features everything you need for a smooth and easy Android penetration testing experience.

Contact us at: appuse@appsec-labs.com

<https://appsec-labs.com/appuse>

Report bug / Request a feature

Bug mail@exmaple.com Submit

Description:

Load APK

The Load APK is the most basic action the dashboard can do. The Load APK button will load an APK into the dashboard (figure 11) from the file system or from the emulator 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. The APK that loads will copy the targeted APK or pull it to the path “/appuse/targets/<apk_name>”

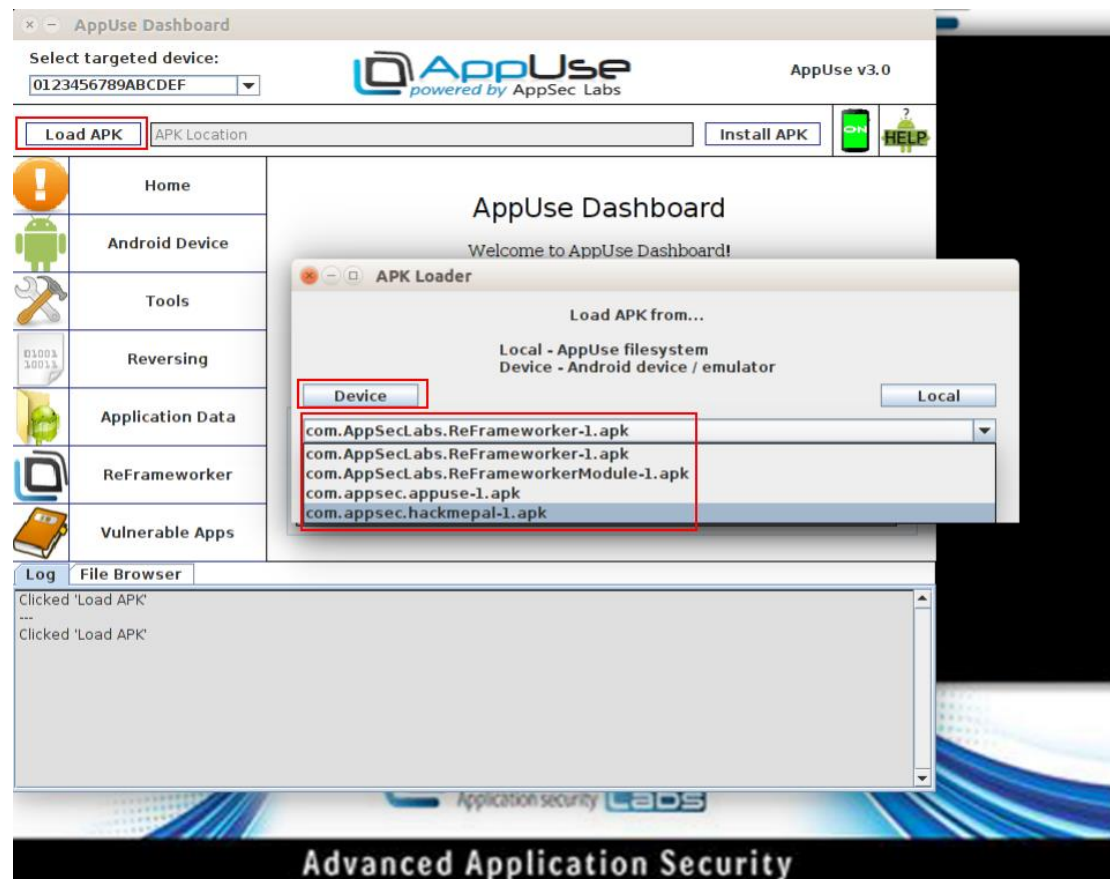


Figure 12: The Load APK action can retrieve APKs from the device or use an existing APK from the file system.

Install APK

The Install APK button will appear after the user has loaded the APK from the file system. This button allows the pentester to install an APK on a running emulator invoked from the Dashboard. In case the application is already installed, the Dashboard will ask the pentester if he would like to reinstall or uninstall and install again the application (see Figure 12).

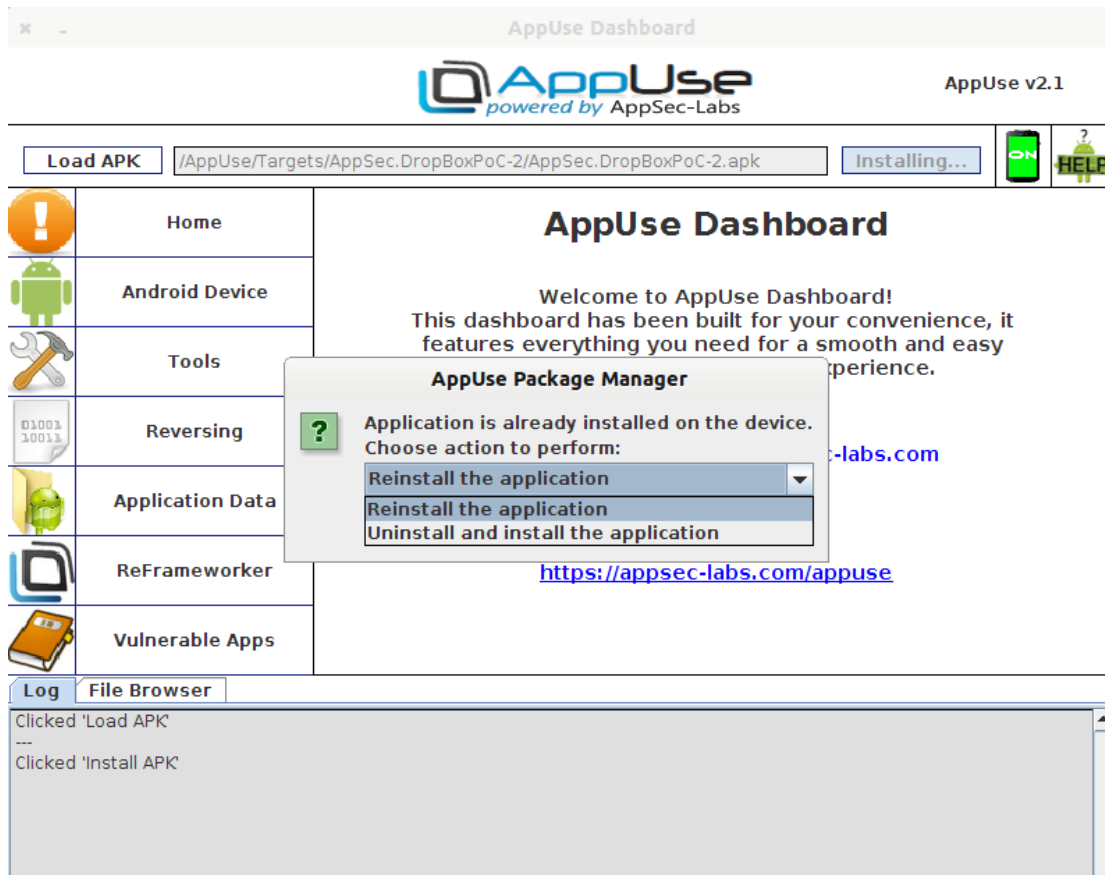


Figure 12: Installing an APK on the emulator via click of a button.

Current Device Status

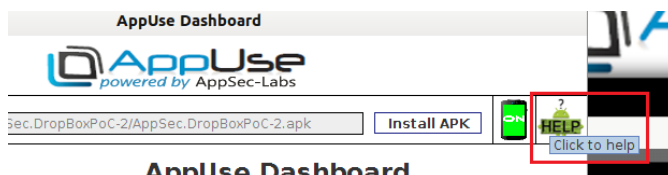
The devices status image shows the current connected device status. It turns to green with the label “ON” while the device is connected to the AppUse VM, and turns to red with the label “OFF” while there is no device connected. It will help you avoid using the “Check Device” button and gain a fast indicator if your device is connected.



Figure 13 – Connected device status

Help

The help image allows you to gain information about the currently selected section (e.g. Reversing). By clicking it, an AppUse user guide of the current section chapter will be opened in the browser, which helps you understand what the purpose is for each button in the dashboard.



Home

The home section was implemented in order to contact with us (AppUse@AppSec-Labs.com) in any issue about AppUse.

Report bug / Request a feature

it is possible to send bugs and request new features via the “Report bug / Request a feature” feature. We will be glad to receive ideas of new features, requests to add you to our mailing list, bugs, etc.

Android Device

The Android sections are implemented in order to perform actions on the emulator device. Here it is possible to perform the following actions by one click:

Launch Emulator

The Launch Emulator button was implemented in order to turn on the AppUse Emulator.

Restart ADB

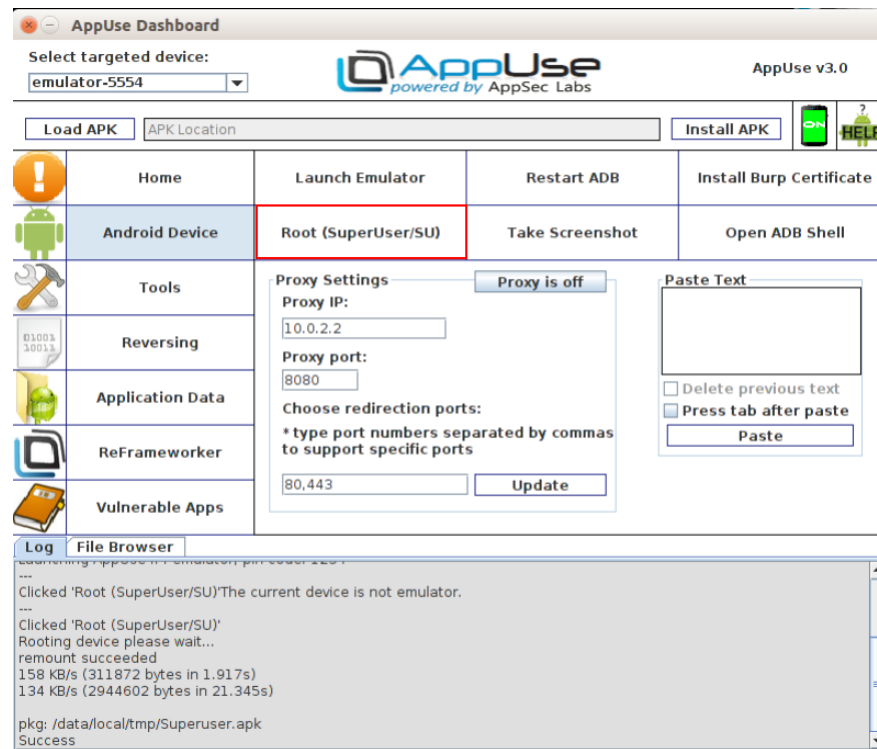
The Restart ADB button was implemented in order to restart the ADB server so AppUse can recognize the device, in case the ADB server is not up and is intended to prevent bugs. In AppUse 1.8 an automatic mechanism was implemented to check if the server is down and restarts the ADB in order to prevent bugs and optimize the pentester’s work.

Install Burp Certificate

The “Install Burp Certificate” button will allow AppUse install the burp certificate in the emulator in order to ease you install the burp certificate. In the next version, it will install it for any device connected to AppUse VM.

Root (SuperUser/SU)

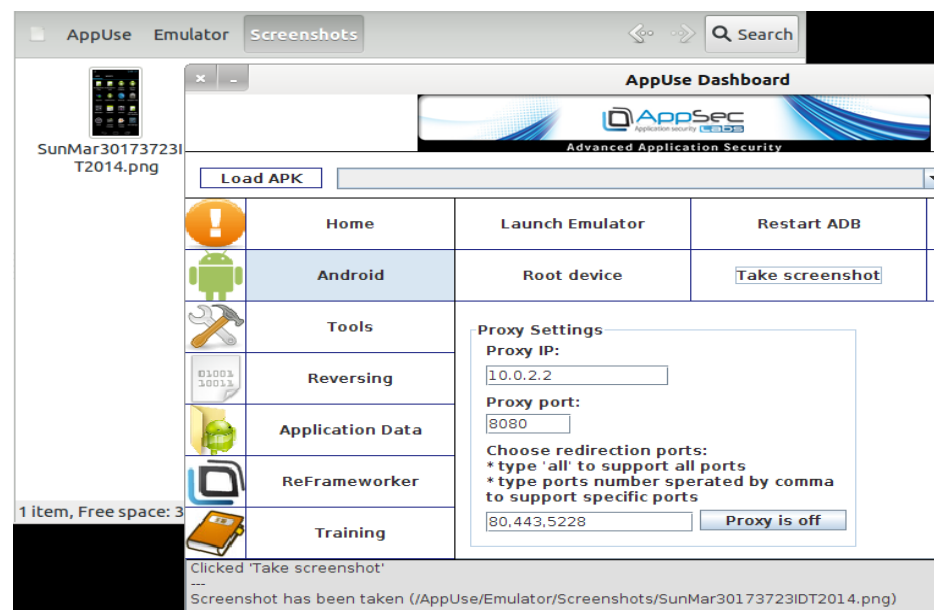
Root privileges on an emulator may come in handy in a penetration test but normally can consume time. The AppUse Dashboard has a built in option to automatically root the emulator with a click of a button and by clicking again the root button it is possible to verify that it is rooted .



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

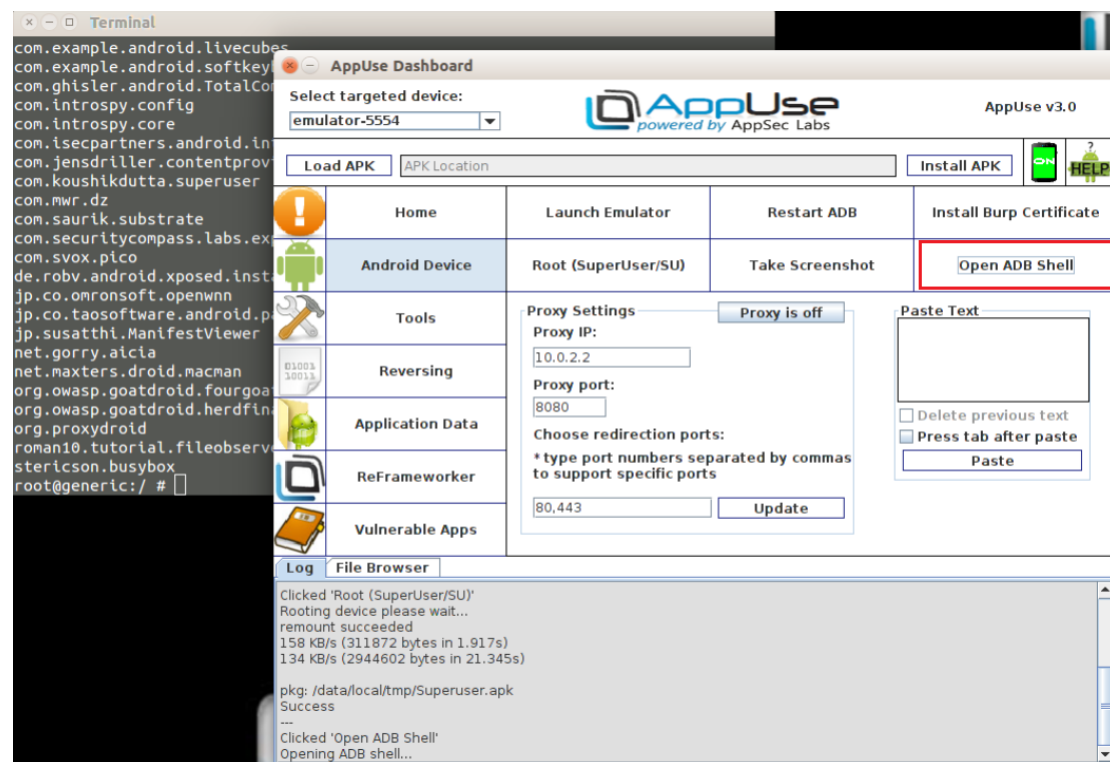
Take Screenshot

The Take Screenshot button was implemented in order to take a screenshot of the emulator. The screenshots will be saved in path /appuse/.emulator/Screenshots/



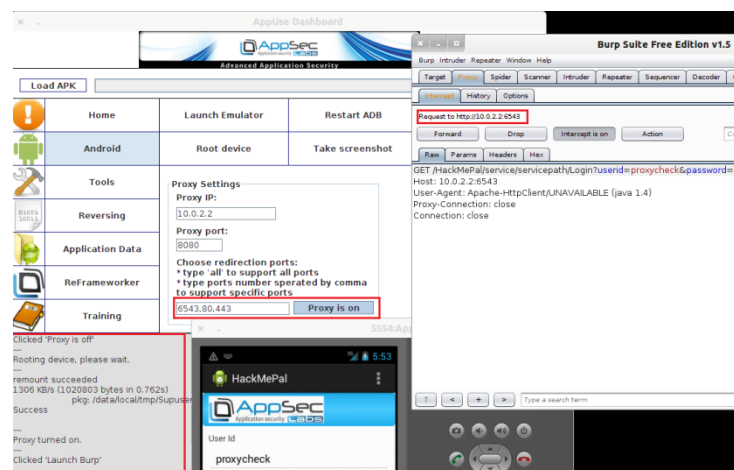
Open ADB shell

The Open ADB shell button was implemented in order to ease the pentester in opening the ADB shell terminal.



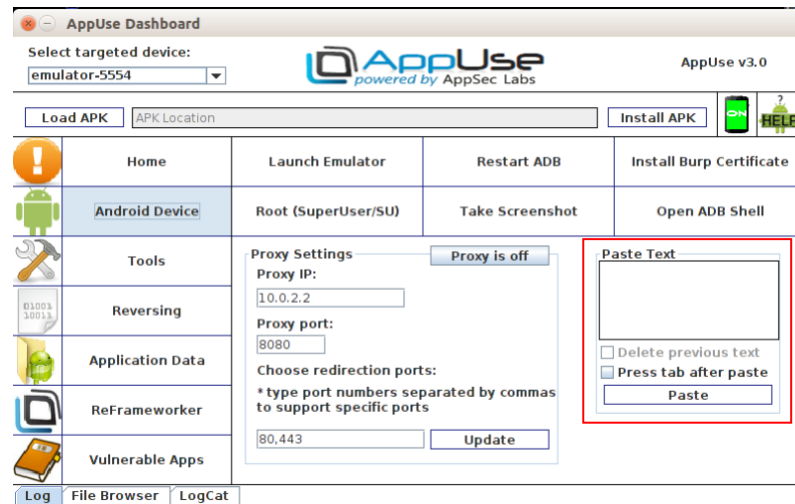
Proxy Settings

The Proxy Settings panel was implemented in order to ease the pentester in rooting the device and mess with proxy applications intent to turn on the proxy on the device. AppUse is preinstalled in the AppUse application which communicates with the dashboard allowing the pentester to turn on /off the proxy in the emulator with a single click. The proxy settings allow the pentester to determine the Proxy IP, Port. In addition, a powerful feature was added which allows the pentester to determine the redirection ports. In case the target application is using the binary protocol, it is possible to set the specific port or choose all ports and intercept the binary protocol's traffic. In case the device isn't rooted, AppUse will root it automatically.



Paste Text

The Paste Text panel designed to solve the problem that it is impossible to paste text directly to the Android device, this feature will allow pentesters easily pasting text from the OS to the Android device.



Tools

The Tools section is implemented in order to give the pentester fast access to useful tools that are used in tests. It is possible to perform the following actions with one click:

Launch Burp

The Launch Burp button is implemented in order to ease the pentester launching burp proxy.

Launch Firefox

The Launch Firefox button is implemented in order to ease the pentester launching Firefox browser.

Launch Wireshark

AppUse comes with the Wireshark sniffer 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 enables the pentester to deeply inspect all the traffic on the device without the limitation of HTTP-based protocols. The Launch Wireshark button is implemented in order to ease the pentester launching Wireshark.

Launch Eclipse

The Launch Eclipse button is implemented in order to ease the pentester launching Eclipse IDE.

Launch NetBeans

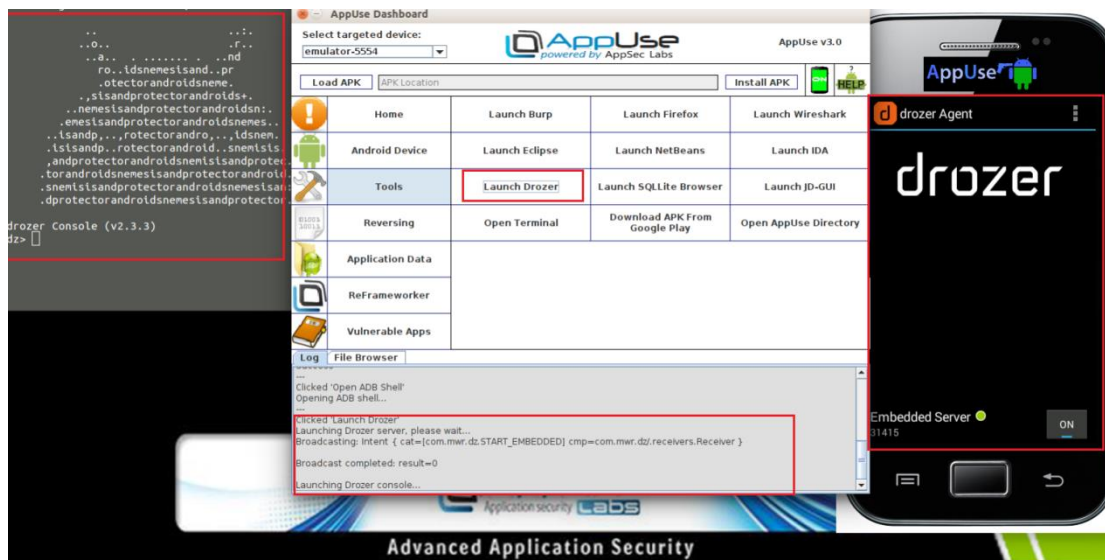
The Launch NetBeans button is implemented in order to ease the pentester launching NetBeans 8.0 which allows you perform debugging to applications.

Launch IDA

The Launch IDA button is implemented in order to ease the pentester launching IDA which allows you perform reverse engineering to binary files (e.g. so files).

Launch Drozer

The Launch Drozer button is implemented in order to ease the pentester launching Drozer console client. After turning on the server in the device via Drozer Agent, the pentester should perform port forwarding, then open the mercury console. This is all performed with a single click of the Launch Drozer" button.



Launch SQLite Browser

The Launch SQLite Browser button is implemented in order to ease the pentester launching the SQLite Browser, in order to edit or view database files.

Launch JD-GUI

The Launch JD-GUI button is implemented in order to ease the pentester launching the JD-GUI, in order to view source code of JAR files.

Open Terminal

The Open Terminal button is implemented in order to ease the pentester launching the shell, in order to perform actions in the system.

Download APK From Google Play

The Download APK From Google Play button is implemented in order to ease the pentester download application from google play, because there is no google play application in the emulator.

Open AppUse Directory

The Open AppUse Directory button is implemented in order to ease the pentester in opening the AppUse directory, in order to view files and perform actions on it.

Reversing

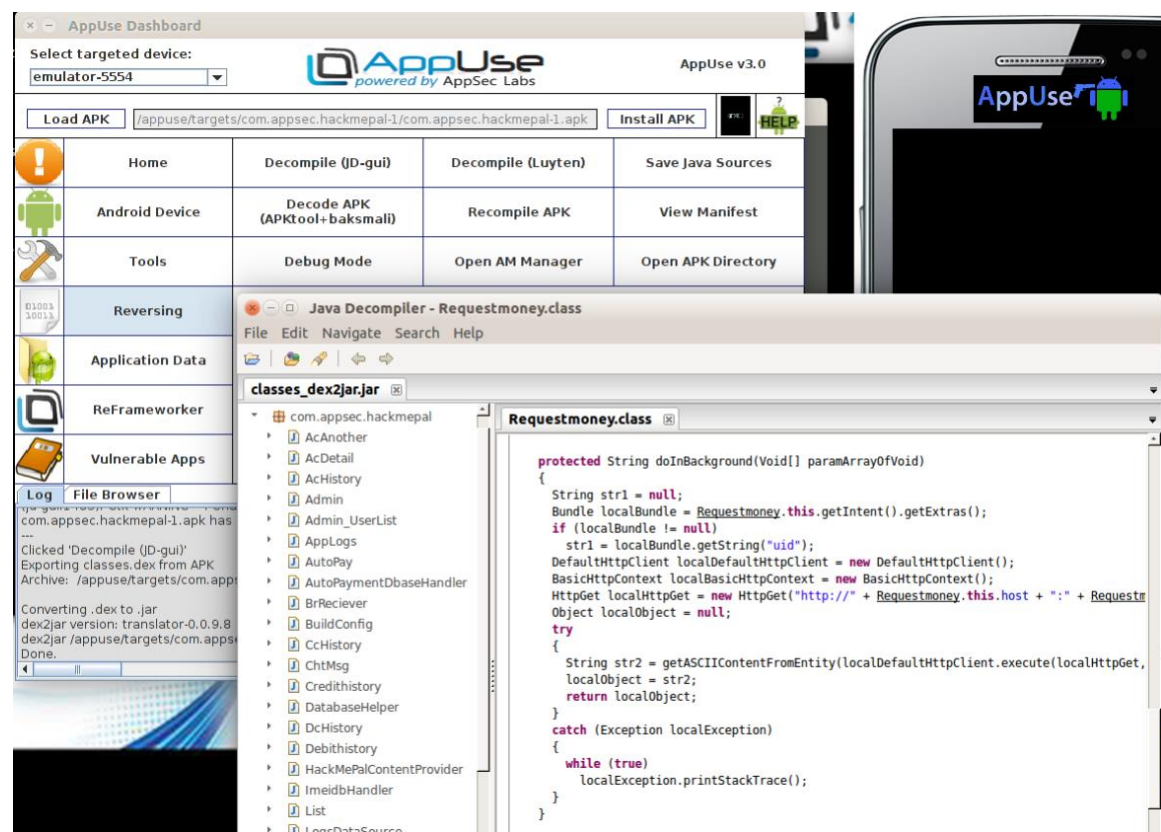
AppUse includes the most advanced tools used to decode and reverse engineer APK's. Once an APK is loaded to the dashboard all the tools are preconfigured to use it and all the tools and frameworks are leveraging the pentester to reach full coverage.

The Reversing section is implemented in order to ease the pentester in performing reversing actions like pulling APK from the device, decamping it, disassembling it, assembling it, converting the APK to debug mode, etc. This section will make the pentester work much faster. It is possible to perform the following actions with one click:

Decompile (JD-GUI)

JD-GUI is a framework aimed to disassemble .jar files. Once a .dex file had been converted to .jar, the 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.

The Decompile button was implemented in order to ease the pentester in decompiling the targeted APK, converting it to a JAR file (via dex2jar) and using JD-GUI to view the source code. This is all performed with a single click. AppUse will pull the targeted APK, decompile it, and open JD-GUI with the targeted APK's JAR.



Decompile (Luyten)

This button does the same operation as the button above but opens the JAR file with Luyten decompiler.

Save Java Sources

The “Save Java Sources” button will save the java sources of the targeted APK into “decompiled” directory in the targeted APK directory.

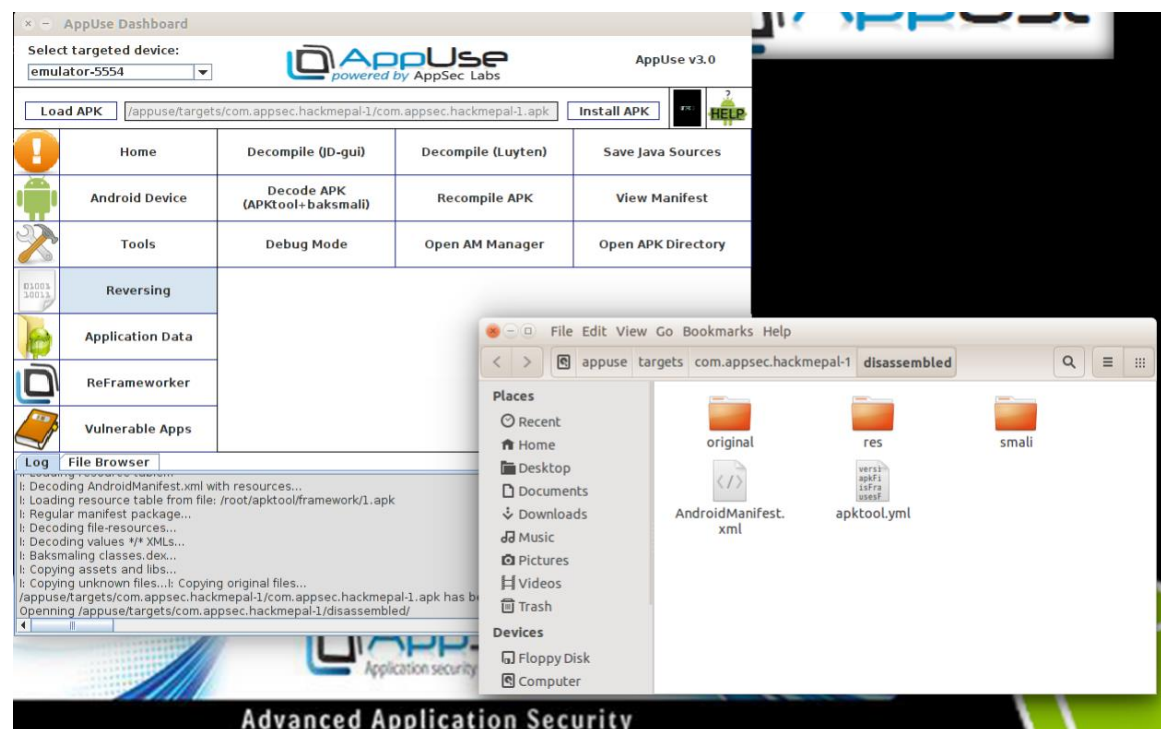
Decode APK (APKtool+baksmali)

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

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.

The Baksmali Dissassemble button was implemented in order to ease the pentester to pull the APK and disassemble the targeted APK via multiple commands. This is all done with a single click.

Once baksmali has been used on an APK, the /appuse/targets/<Targeted APK>/Dissassembled/folder will contain all of its Dalvik assembly code in a human-readable format. The assembly can then be modified and provide added instructions for the pentester.



Recompile APK

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.

The Smali Assemble button was implemented in order to assemble the disassembled folder via the “Baksmali Dissassemble”. After the pentester has modified the Smali code, he will like to convert it to an APK file and install it. This is all done with a single click on the Smali assemble button. AppUse will assemble the “Disassembled” directory and creates a signed APK.

View Manifest

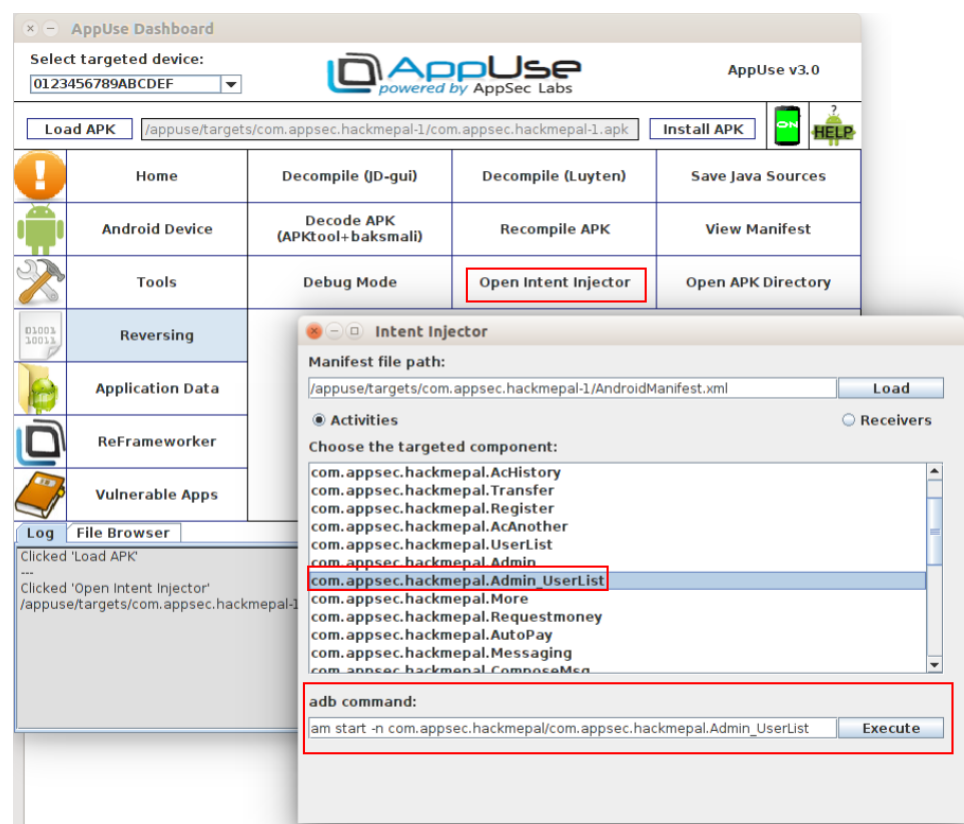
The View Manifest button was implemented in order to ease the pentester to open the targeted APK manifest. It will extract the APK manifest and open it via Firefox.

Debug Mode

The Debug Mode button was implemented in order to ease the pentester in debugging the targeted APK, by converting it automatically into debug mode. Via a single click, the application will disassemble the targeted APK, modify its manifest, assemble it, and sign it.

Open Intent Injector

The Open Intent Injector button was implemented in order to ease the pentester to execute “AM” commands in order to start application’s activities or send a broadcast messages, it will automatically fetch activities and receivers from the manifest of the targeted application, thus, allows the pentester easily choose the specific activity and receiver he wants to test and easily execute AM command from the Intent Injector frame.



Open Target Directory

The Open Target Directory button is implemented in order to ease the pentester to open the targeted application's directory, in order to view files and perform actions on it.

Application Data

The Application Data section is implemented in order to ease the pentester to access the targeted application's files. The "Load APK" button is replaced with "Load Data" button, loads the data applications directories into the list, and the user can use the filter to filter the directory names. The pentester will choose from the list the targeted application's directory and click on the "Load Folder" in order to execute actions on the targeted directory. It is possible to perform the following actions with one click:

View File

The View File button was implemented in order to ease the pentester in viewing files inside the targeted application. After loading the folder into the tree view, it is possible to select a file inside the tree view, click the cat file, and view the content of the selected file.

Edit File

The Edit File button was implemented in order to ease the pentester in modifying files inside the targeted application. With a single click AppUse will pull the file and open it in an Editor (SQLite for DB files and getit to others extension). In order to ease the pentester in modifying the file, after the pentester modifies the file and saves it, it will get pushed back to the device automatically.

Pull File/Folder

The Pull File/Folder button was implemented in order to ease the pentester in pulling files or directories from the targeted application. The tree view functionally allows the pentester to view and choose what files or folders to pull, in a much faster way.

Extract Databases

The Extract Databases button was implemented in order to ease the pentester to view the databases files. Thus, instead of pulling the DB files and opening each one in the SQLLITE browser, with a single click AppUse will pull the DB files and parse the databases data to HTML via doxygen, and allows the pentester to view the entire table in each file.

Open App Data Directory

The Open Target Directory button is implemented in order to ease the pentester in opening the targeted application's directory, in order to view files and perform actions on it.

ReFrameworker

ReFrameworker Dashboard

The ReFrameworker Dashboard button was implemented in order to launch the ReFrameworker platform that will be elaborated later on in this document.

Enable ReFrameworker

The Enable ReFrameworker button was implemented in order to easily launching the Xposed app that should be installed to allow ReFrameworker works properly.

Select Targeted App

Due to changes in version < 3, ReFrameworker runtime manipulation will attach a specific targeted application, so after installing the Xposed, you should choose the specific targeted application you want to manipulate.

Vulnerable Apps

The Training section is implemented in order to turn on the server-side training applications; it is possible to turn on HackMePal HTTP/S servers, GoatDroid and ExploitMe.

The screenshot displays the AppUse Dashboard, a web-based interface for managing applications. The top header includes the 'AppUse Dashboard' title, a 'Select targeted device' dropdown menu with the value '0123456789ABCDEF', the 'AppUse powered by AppSec Labs' logo, and the version 'AppUse v3.0'. Below the header, there are buttons for 'Load APK' and 'Install APK', along with a file path input field containing '/appuse/targets/com.appsec.hackmepal-1/com.appsec.hackmepal-1.apk'. A sidebar on the left contains a list of navigation items: Home, Android Device, Tools, Reversing, Application Data, ReFrameworker, and Vulnerable Apps. The 'Vulnerable Apps' item is highlighted with a red box. The main content area shows a table of vulnerable applications with columns for the application name and its status. The table includes 'HackMePal - Server', 'GoatDroid - Server', 'ExploitMe - HTTP Server', and 'ExploitMe - HTTPS Server'. The 'HackMePal - Server' button is green, indicating it is on, while the others are red, indicating they are off. A 'Section Description' box explains that some vulnerable apps have a server side and can be turned on or off, with the button color indicating the status. At the bottom, a 'Log' section shows a list of actions: 'Clicked \'Load APK\'', 'Clicked \'Open Intent Injector\'', 'Clicked \'HackMePal - Server\'', and 'HackMePal is now listening to port 6543'. The last two log entries are highlighted with a red box.

AppUse Dashboard

Select targeted device: 0123456789ABCDEF

AppUse powered by AppSec Labs

AppUse v3.0

Load APK /appuse/targets/com.appsec.hackmepal-1/com.appsec.hackmepal-1.apk Install APK

Home HackMePal - Server GoatDroid - Server ExploitMe - HTTP Server

Android Device ExploitMe - HTTPS Server

Tools

Reversing

Application Data

ReFrameworker

Vulnerable Apps

Section Description

Some of the vulnerable apps in the emulator have a server side. In this section you can turn their server on or off. The server button is green when the server is on, and red when it is off.

Log File Browser

Clicked 'Load APK'

Clicked 'Open Intent Injector'

/appuse/targets/com.appsec.hackmepal-1/AndroidManifest.xml

Clicked 'HackMePal - Server'

HackMePal is now listening to port 6543

AppUse - 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.

AppUse's heart 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 an 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 keys 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 a 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.

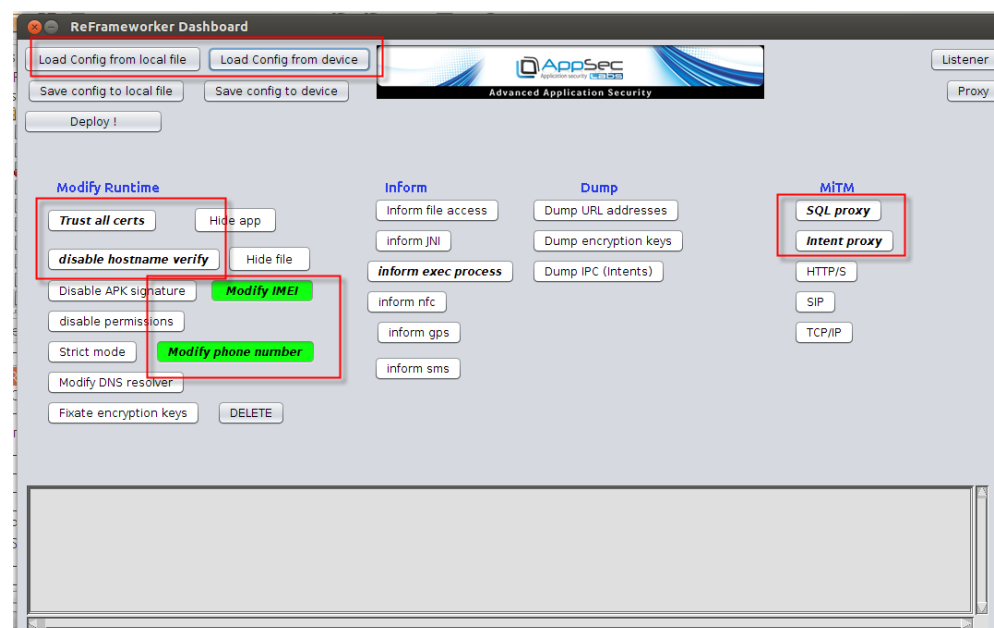


Figure 13: Loading rules from config file.

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, perform some string replacement, send some data to the ReFrameworker dashboard, and so on.

Next, 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.

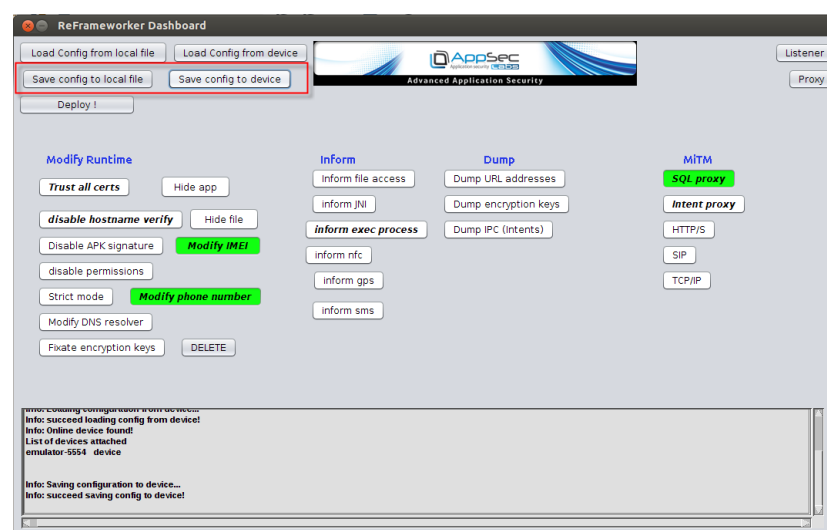


Figure 14: Saving rules to config file.

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

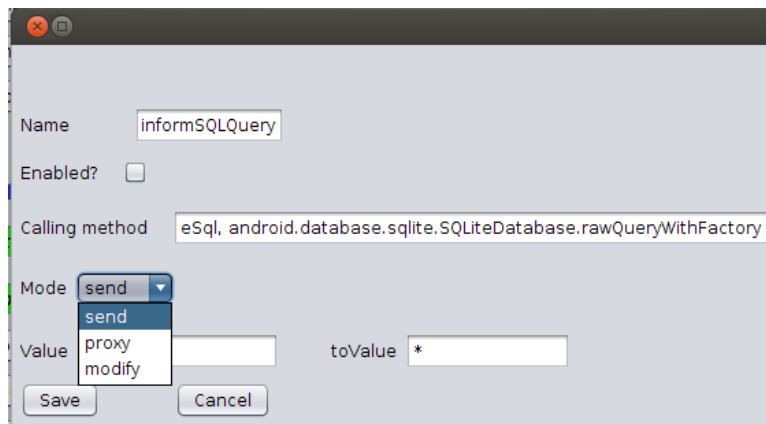


Figure 16: An example for rule configuration.

For example, here's how the config file will look like after generating new rules for the runtime. The following is an example of the "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>333333333333333333333333</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.execSQL, 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, where our hooks will look for it. The location of this file should be at `"/data/system/Reframeworker.xml"` – so that our hooks that 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):

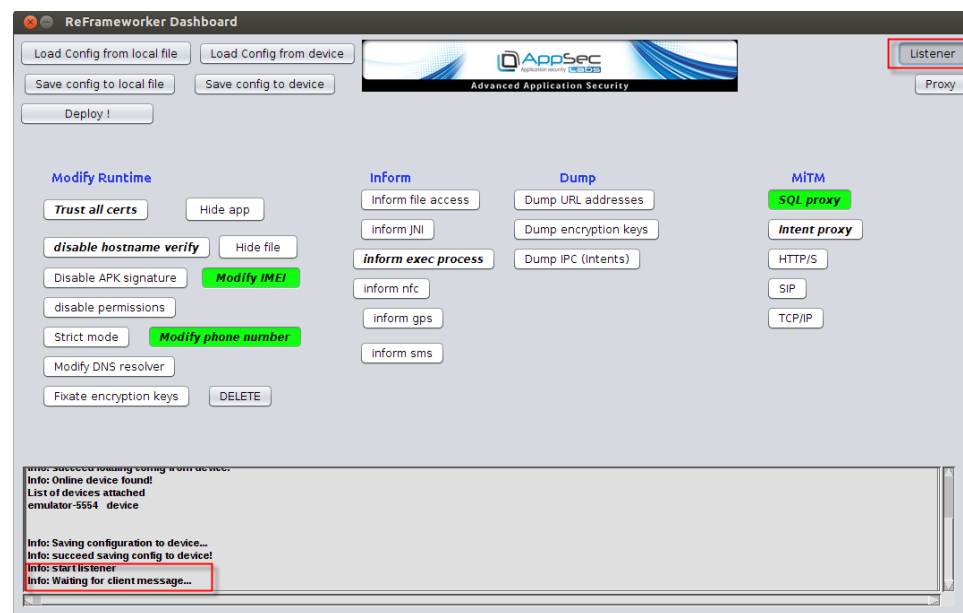


Figure 18: Starting the listener.

Now after starting the listener the dashboard is ready for any incoming messages.

AppUse also has an advanced feature that lets the user intercept some internal information from inside 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 many hooks at some key places. As part of the research, after finding interesting places we want to control, such as handling of files, communication, encryption, etc., we placed calls at those locations to the ReFrameworker controller. The controller's responsibility is to 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 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 in 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 a modified received value.

All it takes at the dashboard's side is to operate the proxy (figure 20).

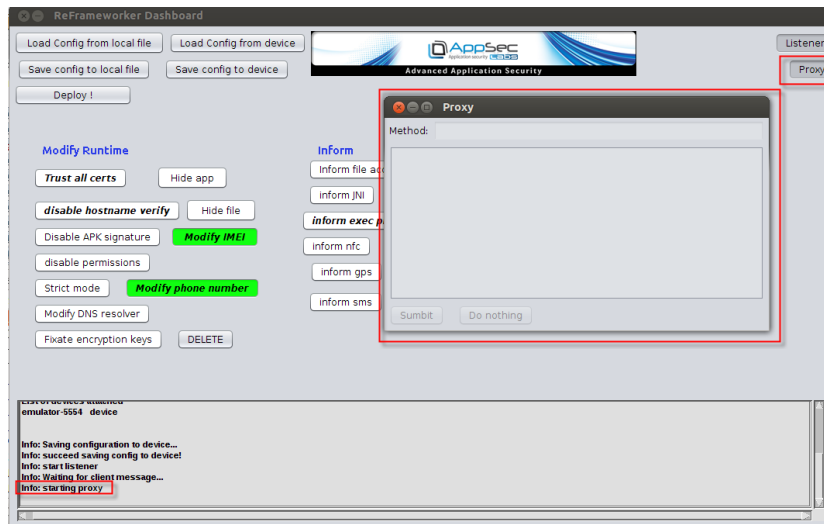


Figure 20: Starting the proxy.

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. It does this while the Android app is waiting for the response! (See figure 21)

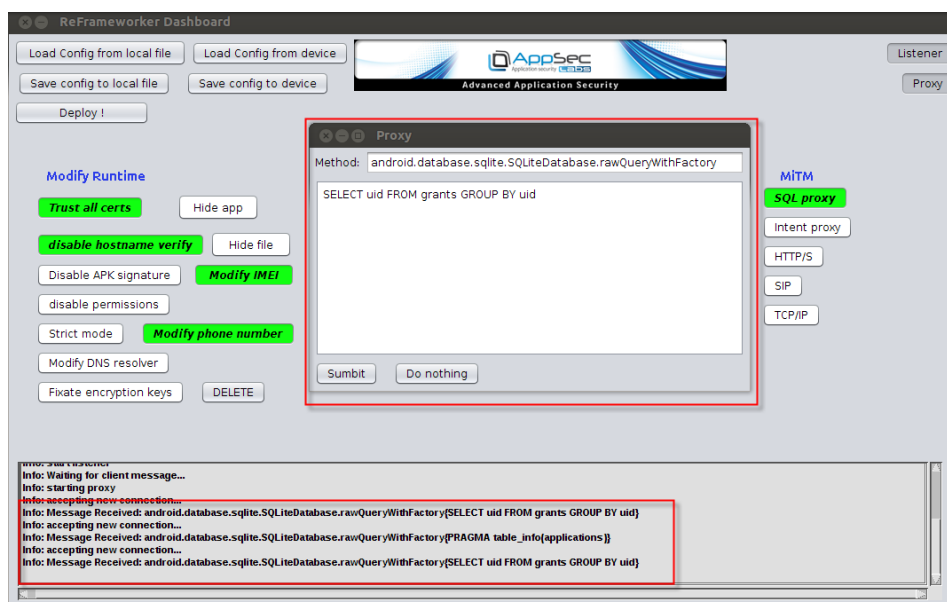


Figure 21: Handling the incoming message using the proxy.

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.

A screenshot of a configuration window with a light gray background. At the top left are standard window control buttons (red, yellow, green). The window contains the following fields: 'Name' with the text 'informExecProcess'; 'Enabled?' with a checked checkbox; 'Calling method' with the text 'java.lang.Runtime.exec'; 'Mode' with a dropdown menu showing 'send'; 'Value' with a text box containing '*'; and 'toValue' with a text box containing '*'. At the bottom are 'Save' and 'Cancel' buttons.

Explanation – the 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. The calling method is set for the relevant hooked method.

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

A screenshot of a configuration window similar to the first one. The 'Name' field contains 'informSQLQuery'. The 'Enabled?' checkbox is checked. The 'Calling method' field contains the text 'android.database.sqlite.SQLiteDatabase.execSQL, android.datab' and is highlighted with a red rectangular border. The 'Mode' dropdown shows 'send'. The 'Value' and 'toValue' fields both contain '*'. 'Save' and 'Cancel' buttons are at the bottom.

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

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

A screenshot of a configuration window titled 'informSQLQuery'. The window has a light gray background and a dark gray title bar with standard window controls. The fields are as follows: 'Name' is 'informSQLQuery', 'Enabled?' is checked, 'Calling method' is 'android.database.sqlite.SQLiteDatabase.executeSql, android.datab', 'Mode' is a dropdown menu set to 'proxy' (highlighted with a red box), 'Value' is '*' and 'toValue' is '*' (both highlighted with a red box). At the bottom are 'Save' and 'Cancel' buttons.

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

Example #4 – Trust all certificates.

A screenshot of a configuration window titled 'trustAllCerts'. The window has a light gray background and a dark gray title bar with standard window controls. The fields are as follows: 'Name' is 'trustAllCerts', 'Enabled?' is checked, 'Calling method' is 'javax.net.ssl.SSLContext.init', 'Mode' is a dropdown menu set to 'modify' (highlighted with a red box), 'Value' is '*' and 'toValue' is 'true' (both highlighted with a red box). At the bottom are 'Save' and 'Cancel' buttons.

Explanation – the mode is set to "modify" since we want to replace the hooked value (specifically, the Boolean value of whether the certificate should be trusted). The 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. The calling method is set for the relevant hooked method.

Example #5 – Disable hostname verification.

A screenshot of a configuration window with a light gray background. At the top, there's a title bar with standard window controls. The main area contains several fields: 'Name' with the value 'noHostnameVerifier', 'Enabled?' with a checked checkbox, and 'Calling method' with the value 'javax.net.ssl.DefaultHostnameVerifier.verify,javax.net.ssl.DefaultHos'. Below these, there's a 'Mode' dropdown menu set to 'modify', which is highlighted with a red rectangle. Underneath the mode dropdown is another red rectangle enclosing two input fields: 'Value' with the asterisk '*' and 'toValue' with the value 'true'. At the bottom, there are 'Save' and 'Cancel' buttons.

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

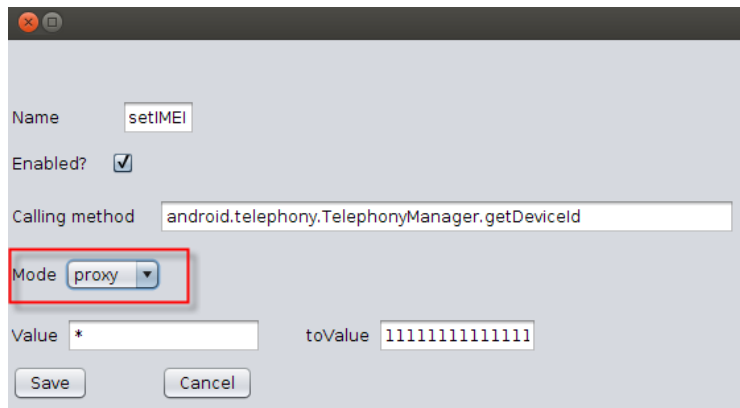
Example #6 – replace the value of the phone IMEI number with another value.

A screenshot of a configuration window similar to the one above. The 'Name' field contains 'setIMEI'. The 'Enabled?' checkbox is checked. The 'Calling method' field contains 'android.telephony.TelephonyManager.getDeviceId'. The 'Mode' dropdown is set to 'modify'. Below it, the 'Value' field contains '*' and the 'toValue' field contains '111111111111', both of which are highlighted with red rectangles. 'Save' and 'Cancel' buttons are at the bottom.

Explanation – the 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. The calling method is set for the relevant hooked method.

Note – if we 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 – the proxy (break and modify) is the value phone IMEI number.



Explanation – the mode is set to "proxy" since we want to modify this data in real-time. Other values have stayed the same (compared to the previous example).

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

Link to AppUse

AppUse can be downloaded from here:

<https://appsec-labs.com/appuse>

iOS Dashboard Structure (Beta Version):

The dashboard is divided into 6 sections; each one is aimed to a specific purpose.

General

iOS dashboard relies on the same concept as Android dashboard which was the first product that was released in Appuse – to simplify the process of application analyzing by grouping all the well-known tools along with AppSec-Labs' tools in one dashboard.

Application list

Application list bar shows the installed applications in the iOS device after making a successful connection, and allows you to choose out of the dropdown list a specific application that you would like to test. Note that after making a successful SSH connection or installation of a new application, you should refresh the list using the cycle icon from the right.

Home

Connecting to the iOS device

Note, the device has to have iNalyzer agent installed inside. In order to install it, browse from inside the iOS device to: <https://appsec-labs.com/cydia> and add AppSec repository and install iNalyzer package.

The first step is to connect to the iOS device in order to get information out of it. In home section, we supplied a quick device configuration tool to open an SSH connection to the device. Once the connection is made, the application list at the top will be updated and we can start working on the tested application.

Analyze Applications

Install Agent

The main activity of the application analyzer works with a well-known tool made by AppSec Labs called "iNalyzer". Since the tool works with an agent which is installed on the jailbroken iPhone as a package, this button helps to speed things up and install it by clicking "Install Agent". Note, Currently the agent supports iOS 8.x and 9.x.

Get information

By clicking get information, AppUse gets from the agent information about the installed applications on the device, for later examination of your choosing.

Get IPA

Get IPA action button copies the IPA file (which is the application itself, "pre-installed" condition) to the local folder on the machine named targets (located on the Desktop)

Get Sandboxed Folders

After installing the IPA on the device, the application is operating within its own container, this container contains the sandboxed directories of the application, that may change according to different actions performed in the application (login, downloading files via the application, caching etc.). By clicking "Get Sandboxed Folders" button AppUse copies the container to the local folder "targets" (contained in Appuse folder on the Desktop) for further examination.

Perform Dynamic Analysis

AppUse asks the iNalyzer agent to perform runtime analysis of the application and then opens the browser with a view of the iNalyzer output.

Perform Static analysis

AppUse asks the iNalyzer agent to perform runtime analysis of the application and then opens the browser with a view of the iNalyzer output where the researcher will be able to examine the application's strings and URL schemes. By using this analysis method, sensitive information such as administrative/Premium endpoint can be revealed, along with hardcoded keys and passwords.

Note, static analysis may take a while. Wait until the browser gets open.

Open Folder

Open's the folder of the target folder of the application analysis.

Console/NSlog

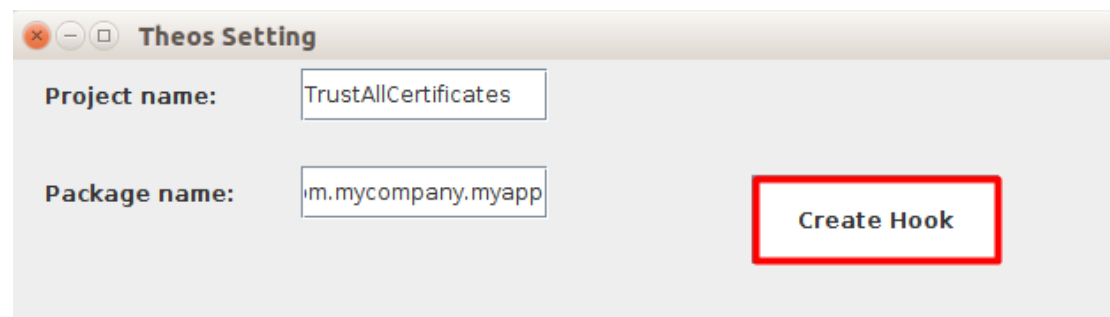
This section includes action buttons to control the NSLog of the iPhone device, using this section the researcher can follow the information printed by the application in the NSLog during legitimate or illegitimate functionality.

Runtime Hooking

This section includes the action buttons to control the "Theos" framework which is responsible for hooking to processes during the application runtime.

Create Hook

By using the hook creation functionality, the researcher can change some of the application's functionality using his own code (hooks). Simply create a hook to use with Theos:



Compile Hook

Compiles the hook.

Install Hook

Installes the hook on the device.

Jailbreak

This section gives the researcher references to "Official" sources which are responsible for new public jailbreaks and additional information about existing jailbreaks for every OS version. Site such as the following are included:

- <https://www.theiphonewiki.com/wiki/Jailbreak>
- <https://canijailbreak.com/>
- <https://support.apple.com/en-us/HT201263>
- <https://ipsw.me/>

Tools

This section is very similar to the tools section in the android dashboard, and can run almost the same set of tools which is used for android application analyzing process:

- BurpSuite Free edition
- Firefox browser
- Wireshark sniffer

- SQLite Browser
- Hopper
- plistutil

iNalyzer agent CLI

Another way to communicate with iNalyzer agent, is to connect to the device using SSH and then run the commands:

```
cd /Applications/iNalyzer.app
```

```
./iNalyzer.sh help
```

In order to get the iNalyzer's CLI help.