MWR INFOSECURITY

Mercury User Guide v1.1

Tyrone Erasmus

2012-09-03
PUBLIC

Index

1. Introduction

This guide explains how to get started with using Mercury. Mercury is a framework that is changing all the time and so this document might not always be 100% accurate for the latest development version on Github. This document will however always be relevant for the latest release version on http://labs.mwrinfosecurity.com/tools/2012/03/16/mercury/

Mercury is an advanced tool that requires experience with Android in order to fully make use of all its functionality. If you haven't spent the required couple of hours learning Android basics, some of the functions and features of Mercury may not make total sense. With that said, there is a fair amount of functionality, especially in the **modules** section that allows users to find new vulnerabilities without much prior knowledge or Android exploitation experience.

2. Getting started

The following explains what is required, where to download and how to get started with setting up the environment needed to run Mercury.

2.1. Recommended requirements

- Phone or emulator running Android 2.2+
- Computer running Linux and Python 2.7. Mercury works on Windows and Mac but the autocompletion of commands is not supported on these platforms due to the reliance on the GNU Readline library.

2.2. Download locations

Release versions of Mercury are stable and released on an ad hoc basis when milestones have been reached. Supporting documentation will be updated and released with every new release version of Mercury. To download the latest release version of Mercury, please visit
http://labs.mwrinfosecurity.com/tools/2012/03/16/mercury/downloads/

Mercury is in constant development and so some may prefer to have the freshest features available from day to day. Get it from Github using:

```
git clone https://github.com/mwrlabs/mercury.git
```

2.3. Setting it up

2.3.1. Recommended tools

It is recommended that basic Android tools are installed on the computer where Mercury will be used. Tools like ADB (Android Debug Bridge) are very useful to have and are required when using Mercury over USB or with an emulator. ADB is available in some software repositories and can be installed using:

```
sudo yum install android-tools
```

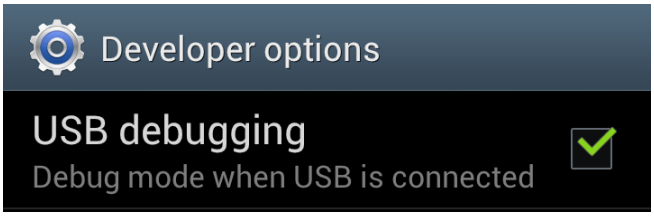
The recommended location to get it from is http://developer.android.com/sdk/installing.html

2.3.2. Android Configuration

Mercury can be used on USB connected Android devices, emulators and over a WIFI connection. The following explains the required setup for each configuration.

USB/Emulator

Make sure that USB Debugging is enabled. This will allow you to use ADB and other development tools on your device.



Connect the USB device or start the emulator and install **mercury-server.apk** using:

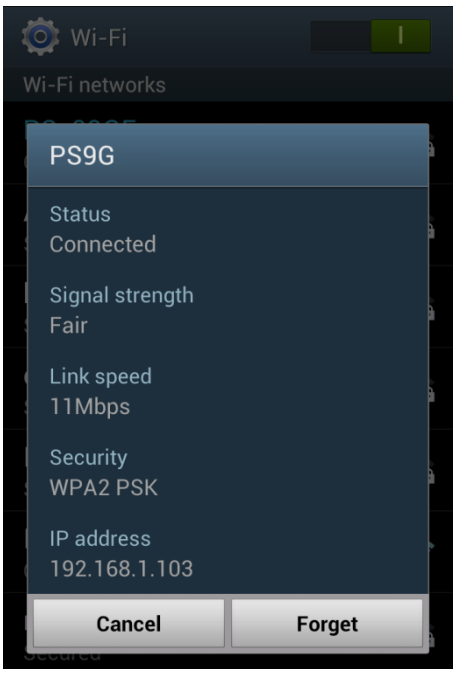
```
adb install mercury-server.apk
```

This is an essential step for any connected device or emulator. Forward the Mercury server port to your localhost so that the client is able to connect to it:

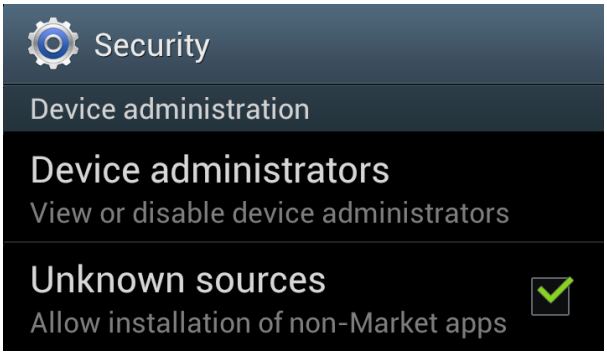
```
adb forward tcp:31415 tcp:31415
```

WIFI

If you are connecting to your device over a wireless network, you will need to get the IP address of your phone. This can be obtained by going to the Wireless settings on the device and clicking on the connected access point. The following dialog will appear containing the device's IP address:



Make sure that "Unknown sources" is checked in the device's security settings, as shown below:



Install **mercury-server.apk** on your device using your favourite method.

3. Using Mercury

3.1. Starting a connection

Open the Mercury app and toggle the On/Off button (Green light = On).

Start the Mercury client using:

```
python mercury.py
```

Type **connect** followed by the IP address of your Android device. The IP address will be 127.0.0.1 if you have a directly connected device or emulator. You should end up with a screen that looks as follows:

If you get a response like ****Network Error** Could not connect to 127.0.0.1:31415** then make sure you have done the following:

- Enabled the Mercury server
- Have access to TCP port 31415 on your Mercury server (remember to forward the port for locally connected devices and emulators).

3.2. First steps

There is inline help available within Mercury's command-line interface. In order to get help about available features, type **?** or **help**. Typing **help (command)** will give an explanation of what the command is and will give examples of typical use-cases. To view command-line usage instructions only for a particular command, type the command appended with a **-h** or **--help**.

If you are familiar with Android development, you will notice the 4 different application components available as sub-folders within Mercury. These application components are:

- Activities – denoted as **activity** in Mercury
- Services – denoted as **service** in Mercury
- Content providers – denoted as **provider** in Mercury
- Broadcast receivers – denoted as **broadcast** in Mercury

Another folder that provides some general tools pertaining to installed Android packages on the connected device is **packages**. Commands inside **packages** can be used to get information about installed packages and view their general attack surface.

A general workflow for examining a single application on the device is as follows:

- Get information about the package of interest
- Examine the attack surface of the package
- Interact with each available application features to find potential problems

To find information about packages of interest, navigate to the **packages** folder. Here are some things to try:

- Type **info** to get information about all the installed packages on the device.
- Try to use **info -f browser** to filter for specific package information about the Android browser
- Look for packages that use the *INTERNET* permission by typing **info -p internet**
- View how you can interact with one of these applications by issuing the **attacksurface (packageName)** command

The following is an example of the **attacksurface** command being issued on an application:

```
*mercury#packages> attacksurface com.android.providers.contacts

0 activities exported
3 broadcast receivers exported
4 content providers exported
0 services exported

shared user-id = 10020 (android.uid.shared)
```

At this point the various application components can be investigated by using the **info -f (packageName)** command in the respective application component folders e.g. **provider** folder

A feature that might appeal to the more advanced users is the **manifest** command in the **packages** section. This essentially replaces the need for using AAPT in order to get the AndroidManifest.xml of a package. An example of viewing Mercury's manifest is shown below:

```
*mercury#packages> manifest com.mwr.mercury -o /home/tyrone/Desktop/AM.xml

<manifest versionCode="2" versionName="1.1" package="com.mwr.mercury">
<uses-sdk minSdkVersion="8" targetSdkVersion="4">
</uses-sdk>
<uses-permission name="android.permission.INTERNET">
</uses-permission>
<application label="@2130968576" icon="@2130837504" debuggable="true">
<activity label="@2130968576" name=".Main">
<intent-filter>
<action name="android.intent.action.MAIN">
</action>
<category name="android.intent.category.LAUNCHER">
</category>
</intent-filter>
</activity>
<service name=".ServerService" enabled="true">
</service>
</application>
</manifest>
```

3.3. Interacting with application components

In order to understand this section, you will need to be relatively familiar with Android development or at least how the Android model works with the various application components.

3.3.1. Intents

Intents are the basis of communication between Android application components that gives Android its modular unique design. To read more about forming different intents in order to interact with application components, please refer to http://developer.android.com/guide/topics/intents/intents-filters.html

3.3.2. Activities

Getting the launch intent of an application can be done using the **launchintent** command. The following will find the launch intent for Google Play.

```
launchintent com.android.vending
```

Rebuilding this intent with the **start** command will launch Google Play. This can be done as follows:

```
start --action android.intent.action.MAIN --category android.intent.category.LAUNCHER --
flags 0x10000000 --component com.android.vending com.android.vending.AssetBrowserActivity
```

When starting new activities, a common error is not specifying the FLAG_ACTIVITY_NEW_TASK flag with the **start** command. The hex representation of it is 0x10000000 as seen in the above command.

An indication that you have forgotten this flag is the following error message:

```
Calling startActivity() from outside of an Activity  context requires the
FLAG_ACTIVITY_NEW_TASK flag. Is this really what you want?
```

When typing an action like **android.intent.action.MAIN** or a category, try to tab complete it to avoid having to type too much. This feature is only supported on Unix-based systems.

Typing **info** in this section will give information about all the exported activities available on the device. Here is an example of starting an activity using the **--component** argument of the **start** command:

```
start --component com.android.browser com.android.browser.BrowserActivity -flags
0x10000000
```

3.3.3. Services

Starting and stopping of services can be performed in Mercury by building intents in a similar fashion to activities. Specifying a component is normally the preferred way of starting and stopping services, which can be done with the **--component** flag of the **start** and **stop** commands. Binding to services is an advanced feature that can only be done using Mercury's reflection interface, which is a topic that is not covered in this guide. More information about this can be found at https://github.com/mwrlabs/mercury/wiki/Using-Reflection.

3.3.4. Content Providers

Content providers are often a source of data leakage in applications and Mercury has some powerful features that allow you to find these vulnerabilities. Navigate to the **provider** section to access these tools. Typing **info** in this section will reveal all of the exported content providers on the device. Some may require certain permissions in order to read or write to them and some don't require any permissions whatsoever.

If you are interested in finding content providers with no required permissions, type:

```
info -p null
```

In order to query a content provider, the following structure general applies:

```
query content://app_authority/table
```

Often, data can be retrieved simply by querying content://app_authority. There is no defined way to find valid content URIs but Mercury has functionality available to try and find valid content URIs in the application's DEX file, which is essentially its executable. Use **finduri (packageName)** in order to find referenced content URIs. An example of querying the settings provider is shown below:

```
*mercury#provider> query content://settings/secure

_id | name | value
.....

5 | assisted_gps_enabled | 1

6 | location_pdr_enabled | 0

7 | network_preference | 1

8 | usb_mass_storage_enabled | 1
```

There are many vulnerabilities to be found in content providers, such as SQL injection, directory traversal and the consequences of manipulating stored data of other applications. For more information about these attacks please see http://labs.mwrinfosecurity.com/assets/246/bheu12-tyrone_erasmus-the_heavy-metal_that_poisoned_the_droid.pdf

Various other SQL-like functions have been provided in this section in order to insert, delete and update rows in a content provider. Familiarity with SQL does help when attempting to exploit a content provider.

3.3.5. Broadcast Receivers

Information about registered broadcast receivers can be found using the **info** feature in the **broadcast** subsection. Looking for broadcast receivers which do not enforce the sending application to have any permissions and examining what actions are performed when a broadcast is sent to the application is the main goal of assessing these application components. Mercury allows a user to send broadcasts to the entire device or to specific applications only, controlled by the intent that is built using the **send** command. To understand more about the broadcast receiver that you are targeting, you would need to decompile the application and check what kind of extras it is expecting and what code is executed once a broadcast is received.

3.4. Shells and Busybox

Mercury has 2 kinds of shell types available in the **shell** folder. Oneoff shell is well-suited for performing tasks on the device that do not require a persistent state. This shell does not actually maintain a persistent shell connection, only opening a new shell, executing the supplied commands and returning the value.

The persistent shell can be used for tasks that require persistence, such as opening a new shell session with elevated privileges. If you can exploit a vulnerability to obtain a root shell, this is the shell you want to be using.

As an example of the difference between the 2 shells, when running Mercury on a rooted/jailbroken device, typing **su** in the persistent shell will give you a root shell whereas the oneoff shell will not.

Busybox is an awesome tool that provides many utilities that Android is lacking. The normal usage of Busybox requires root access on Android but using it inside Mercury does not. In order to upload Busybox, navigate to the **modules** folder and type the following:

```
*mercury#modules> run setup.busybox

[*] Uploading busybox
[+] Succeeded
[*] chmod 770 busybox

You are now able to use $BB to reference the busybox binary inside the oneoff and
persistent shells
```

Once this has been done, Busybox is uploaded into Mercury's data directory and can be used from either of the 2 Mercury shells.

To use Busybox from either shell, type **$BB** as a shortcut to its full path, which is /data/data/com.mwr.mercury/busybox. As an example, type the following to use Busybox's **date** command:

```
oneoffshell:/data/data/com.mwr.mercury$ $BB date
Fri Jul 27 14:42:52 UTC 2012
```

3.5. Downloads and Uploads

Mercury includes functionality to download and upload files between the user's computer and the Android device. Downloading and uploading of files is obviously restricted to the permissions of Mercury e.g. uploading a file to the SD card will not work because Mercury does not have the *WRITE_EXTERNAL_STORAGE* permission. There are few folders on an Android device that can be written to with such limited permissions and that is why the majority of the time uploading files to Mercury's private data folder is advised.

Upload and download functionality can be found inside the **tools** folder in Mercury. The usage of these tools is fairly self-explanatory but **help download** and **help upload** can be used for more information. A common use of the download feature is downloading application APK files. Their location can be found by using the **info** command in the **packages** folder.

3.6. Modules

The **modules** section is where all the automagic happens. Anything from vulnerability scanners, to root exploits, to information pilfering modules can be found here.

To get a list of all the currently available modules, type **list**. You will notice that they have a folder-like structure that have a correlation to where they are found in the *merc/modules* folder in the Mercury folder on your hard drive. Information about each module can be obtained using the **info (module)** command.

Here are a few good reasons to make a Mercury module:

- You find yourself repeatedly performing the same task in Mercury
- You have an automated exploit of some kind that you would like to share
- You have a useful addon that does not quite fit into the everyday usage of Mercury

This is the part where if you are interested further in making your own modules, then you should visit the developer documentation at https://github.com/mwrlabs/mercury/wiki

The modules section is essentially open ended, with the possibility to extend Mercury to do very many things. Especially with the addition of the reflection interface that allows you to dynamically execute code on the Android

device by pushing it from the Python client. More about this can be found in the developer documentation on Github.

4. Want to know more?

For more information about the technical workings of Mercury, see
https://github.com/mwrlabs/mercury/wiki/Developer-documentation

Everyone is invited to come join the Mercury development team on Github at
https://github.com/mwrlabs/mercury/issues for opinions on new features, code contributions and other questions.

Send us a tweet: @droidhg
Email us: mercury[at]mwrinfosecurity.com