Android Memory Dump Analysis

Student Research Paper

submitted: May 2012
by: Simon Leppert
Advisor: Michael Spreizenbarth

Department of Computer Science
Friedrich-Alexander-University Erlangen-Nuremberg
D – 91058 Erlangen
Internet: http://www1.informatik.uni-erlangen.de
Table of Contents

Listings ......................................................... iv

List of Figures ................................................... v

List of Tables ................................................... vi

1. Introduction ....................................................... 1
   1.1. Motivation .................................................... 1
   1.2. Aim of Thesis ............................................... 3
   1.3. Methods of Research ........................................ 4

2. Heap-Dump Acquisition .......................................... 5
   2.1. Basics about the Dalvik Virtual Machine Memory Usage .... 6
   2.2. Acquisition of Heap-Dumps till Android 2.2 ................... 8
   2.3. Acquisition of Heap-Dumps since Android 2.3 ................. 14
   2.4. Heap-Dump Conversion ....................................... 16
   2.5. Heap-Dump Acquisition Python-Script ........................ 17

3. Analysis of the acquired Heap-Dumps .......................... 19
   3.1. Explanation of the MAT-Tool ................................ 20
   3.2. Memory Analysis of the most widely used Android-Apps .... 21
      3.2.1. Facebook-App ........................................... 21
      3.2.2. Twitter-App ............................................. 24
      3.2.3. GoogleMail-App ......................................... 27

4. Conclusion and Further Work .................................. 29
   4.1. Summary ..................................................... 29
   4.2. Limitations .................................................. 29
   4.3. Future Work ................................................ 30
   4.4. Conclusion .................................................. 30
Appendix .................................................................................................................. 34

A. How to get temporary root privileges ................................................................. 35
   A.1. Download and Extract the rageagainstthecage-arm5.bin-exploit ............... 35
   A.2. Activate USB-Debugging on your target phone ........................................... 35
   A.3. Start the ADB-Server and copy the exploit to the phone ......................... 36
   A.4. Run the exploit on your phone .................................................................. 36
   A.5. Check whether you have gained root privileges ........................................ 36
Listings

2.1. Checking which devices are connected to the ADB ............... 8
2.2. Listing the running processes with the ps-command .......... 9
2.3. Setting the needed permissions for /data/misc ................ 10
2.4. Killing the process with the kill-command ................... 10
2.5. Checking if the dump is generated correctly via logcat ....... 11
2.6. Checking if no hprof-file is generated via logcat ............ 11
2.7. Listing /data/misc with ls -l .................................. 12
2.8. Pulling the dump-file to the local machine ................... 13
2.9. Merging the generated dump-parts ............................. 13
2.10. Converting the dump ............................................. 16
2.11. Source: dumpacquisition.py ................................. 17

3.1. OQL-Search for the string "xxx" .............................. 20
3.2. Search for the string "pwd" in a dump of the Facebook-App .... 21
3.3. Email-regex to extract all email-addresses from a txt-file .... 22
3.4. Source: emailaddress.py ........................................ 23
3.5. Search for the username in the Twitter-App .................... 24
3.6. Search for a Tweet in the Twitter-App ......................... 25
3.7. Search for the OAuth-token in the Twitter-App ................. 25
3.8. Search for the username in the GoogleMail-App ............... 27
3.9. Search for the in-box of the GoogleMail-App ................. 28
## List of Figures

1.1. App-downloads and mobile internet-use .......................... 1
1.2. World-wide Smartphone Molbie OS Marketshare ..................... 2

2.1. Memory usage of an Android device ................................. 6
2.2. DDMS-screenshot for hprof-dumping ............................... 15
List of Tables

2.1. Android Distribution March 5, 2012 7
1. Introduction

1.1. Motivation

Analysis of data stored on mobile phones gets more important with their increase of use. Especially in criminal investigations the acquired information could be very useful. Over the past years the use of mobile internet with smartphones is heavily increasing as well as the number of "app"-downloads. Apps are custom-built software-programs for smartphones and tablet computers. Figure 1.1 shows the increasing use of mobile internet in Germany (4) and the number of app-downloads (5).

![Figure 1.1.: App-Downloads and mobile internet-use](image)

The information that is stored on mobile phones, like images, videos, audio recordings and communication patterns could be used as evidence in criminal investigations to prove or disprove theories and presumptions. The common way to acquire such information from a mobile phone concentrates on nonvolatile (NAND flash) memory including steps like "pulling the plug", acquiring the non-volatile data and analyzing it in a forensic manner. Such a procedure prevents from changes to the potential evidence and is accepted by the law. However,
such a protection against changes could cause the loss of important volatile data. Volatile memory can contain the following information:

- Usernames and passwords
- App data
- Encryption keys

Therefore a live-analysis of the volatile memory can be very evident, especially with the increasing information-exchange through the use of social-networking-apps. Through the high possibility of acquiring login-data of social-networking- and web2.0-websites like facebook or twitter, a forensic investigator could easily get access to accounts of a suspicious person.

In this thesis we concentrate on the Android-OS. The world-wide smartphone mobile OS marketshare for Android has increased heavily over the past years. Figure 1.2 shows that in the year 2011 Android passes the iOS and becomes first place of installed mobile-OS (23). In the beginning of the year 2012 nearly every second mobile phone runs under Android with its different versions.

![Figure 1.2.: World-wide Smartphone Molbie OS Marketshare](image-url)
1.2. Aim of Thesis

The aim of this thesis has changed over the course of the year. Originally, the thesis was based on developing a python-script for acquisition and analysis of a heap-dump from Android. However, the way of generating a heap-dump of a running process has changed with the development of Android. From the first version till the 2.2-"Froyo"-version there was the possibility to use the command "kill -10 pid" to automatically generate a heap-dump. In the 2.3-"Gingerbread"-version that is not implemented anymore (22). To acquire a heap-dump with the newer Android versions from 2.3 to 4.0 the Android Developer Team provides us with a tool named Dalvik Debug Monitor Server (DDMS) (2). Hence, the focus of the thesis has shifted from the development of a python script to describing the old and the new way of acquiring a heap-dump of an app and moreover the memory analysis of the most widely used Android apps and the possible forensic relevant results.
1.3. Methods of Research

Within this Student Research Paper we will start explaining some basics about the Dalvik Virtual Machine memory usage in Section 2.1. After that we will show the way of acquiring a heap-dump of a process from a smartphone running Android till version 2.2 in Section 2.2. Thereafter we will explain the same procedure for smartphones running Android 2.3 or higher in Section 2.3. We will utilize widely used tools that are part of the AndroidSDK to achieve these goals. In Section 2.4 we will explain why and how to convert the acquired heap-dumps to a format that is suitable for heap-dump profiling tools. At the end of Chapter 2 we present a script that will automate the procedure of Section 2.2 and 2.4.

In Section 3 we will analyze heap-dumps of three of the most widely used Android-apps in a forensic manner by using profiling tools like the Eclipse Memory Analyzer (MAT) (7) or Visualvm (14) and showing some major security-holes that are present in these apps.
2. Heap-Dump Acquisition

In the next sections we assume that the Android-SDK is installed correctly, either on a windows-system or linux-system. Thereby, tools like Android Debug Bridge (ADB)(9), DDMS (2) and hprof-conv (3) are usually available in the tools- or platform-tools-directory. In Section 2.1 we explain some basics about the Dalvik Virtual Machine memory usage. In Section 2.2 we describe the way of generating a heap-dump for Android 1.0 till 2.2 and transferring it to our local machine. In Section 2.3 we depict the way of acquiring heap-dumps for Android 2.3 till Android 4.0 by using the DDMS-tool that has been available since Android 2.0. In order to analyze such a heap-dump, we need to convert the hprof-format into another hprof-format that is suitable for profiling-tools like MAT or VisualVM (14) that is part of the latest released java-development-kit (jdk) in version 6 update 30. We characterize that procedure in Section 2.4. The following examples were generated on a windows-system, but in Section 2.5 we will provide you with a python-script, that automates the whole process for linux.
2.1. Basics about the Dalvik Virtual Machine Memory Usage

On Android, every application runs in its own process, each of which hosts its own virtual machine (VM). When an application on Android is started, the Dalvik VM gets some memory from the Operating System. The Dalvik VM uses this memory for all its need and part of this memory is called heap memory (15). The Dalvik heap is preloaded with classes and data by a process called ”Zygote” (8) (loading over 1900 classes as of Android version 2.2) (21), but we will explain that process more precisely in Section 2.3. The Dalvik VM, like virtual machines for many other languages, does garbage collection on the heap 1. There appears to be a separate thread (called the HeapWorker) in each VM process that performs the garbage collection actions. In Figure 2.1 you can see an example of the memory usage of an Android device.

Figure 2.1.: Memory usage of an Android device

In contrast to a main memory analysis that would include everything that is shown in Figure 2.1, we will only analyze the heap of a given process, for example

---

1Garbage Collection (GC) is a form of automatic memory management that attempts to reclaim garbage, or memory occupied by objects that are no longer in use by the process
2.1. Basics about the Dalvik Virtual Machine Memory Usage

the "com.facebook.katana"-process that represents the Facebook-App. With the continuous development of the Android-OS we depicted in Section 1.2 that some necessary commands to acquire a heap-dump are no longer usable since Android version 2.3 but the garbage collection is still done, but without the hprof-heap-dump.

Some might say, that we should concentrate on the newer versions of the Android-OS but as you can see in Table 2.1, the older versions are still very commonly used (1). Therefore we will explain the old and the new way of acquiring heap-dumps in order to give a forensic investigator the knowledge to get heap-dumps from every smartphone running an Android-OS.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Codename</th>
<th>API-Level</th>
<th>Distribution in percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Android 1.5</td>
<td>Cupcake</td>
<td>3</td>
<td>0.4</td>
</tr>
<tr>
<td>Android 1.6</td>
<td>Donut</td>
<td>4</td>
<td>0.8</td>
</tr>
<tr>
<td>Android 2.1</td>
<td>Eclair</td>
<td>7</td>
<td>6.6</td>
</tr>
<tr>
<td>Android 2.2</td>
<td>Froyo</td>
<td>8</td>
<td>25.3</td>
</tr>
<tr>
<td>Android 2.3 bis 2.3.2</td>
<td>Gingerbread</td>
<td>9</td>
<td>0.5</td>
</tr>
<tr>
<td>Android 2.3.2 bis 2.3.7</td>
<td>Gingerbread</td>
<td>10</td>
<td>61.5</td>
</tr>
<tr>
<td>Android 3.0</td>
<td>Honeycomb</td>
<td>11</td>
<td>0.1</td>
</tr>
<tr>
<td>Android 3.1</td>
<td>Honeycomb</td>
<td>12</td>
<td>1.1</td>
</tr>
<tr>
<td>Android 3.2</td>
<td>Honeycomb</td>
<td>13</td>
<td>2.1</td>
</tr>
<tr>
<td>Android 4.0 bis 4.0.2</td>
<td>Ice Cream Sandwich</td>
<td>14</td>
<td>0.4</td>
</tr>
<tr>
<td>Android 4.0.3</td>
<td>Ice Cream Sandwich</td>
<td>15</td>
<td>1.2</td>
</tr>
</tbody>
</table>

Table 2.1.: Android Distribution March 5, 2012
2.2. Acquisition of Heap-Dumps till Android 2.2

This section will showcase the way of generating heap-dumps for mobile phones running Android 1.0 till 2.2 (6) (12). We will work essentially with the ADB and operate with a shell on the target-phone. Basically, the ADB provides a terminal-based interface for interacting with your phone’s file system. When you launch the ADB shell, you are dropped into a shell console on the running phone with standard user privileges. The ADB shell is an ash \textsuperscript{2} shell with limited binary tool availability. For example, there are no copy or find commands. There is no inline command editing or shell history. Your limited privileges prevent you from accessing the /data and /cache directories which contain the bulk of the user data you are likely seeking (16).

First of all we checked which devices are connected to the ADB-server. In Listing 2.1 you can see that we ran an emulator but any other smartphone running Android till version 2.2 can be used too.

D:\AndroidSDK\tools>adb devices
List of devices attached
emulator−5554 device

Listing 2.1: Checking which devices are connected to the ADB

As shown in the Listing above, only the running emulator is connected.

\textsuperscript{2}The Almquist shell, also known as A Shell, ash and sh, is a fast, small, POSIX-compatible Unix shell
2.2. Acquisition of Heap-Dumps till Android 2.2

Following that we open a shell on the target. To get the process-ID (PID) of the running process that we want to dump, we use the ps-command to list all active system- and app-processes. The listing below displays only a small portion of the processes that are running on the target-phone.

D:\AndroidSDK\tools>adb shell
# ps
ps
USER PID PPID VSIZE RSS WCHAN PC NAME

root 14 2 0 c00487a4 00000000 S aio/0
root 21 2 0 c01723f4 00000000 S mtdblockd
system 52 30 200868 30200 ffffffff afe0c51c S system_server
radio 93 30 111324 22384 ffffffff afe0d4a4 S com.android.phone
app_1 97 30 121564 30380 ffffffff afe0d4a4 S android.process.acore
app_21 120 30 109636 17956 ffffffff afe0d4a4 S com.android.mms
app_17 137 30 98128 17268 ffffffff afe0d4a4 S com.android.alarmclock
app_11 180 30 132256 36004 ffffffff afe0d4a4 S com.android.browser

Listing 2.2: Listing the running processes with the ps-command
As we already depicted, the shell runs with standard user privileges but in the
next step we assume that our shell has root privileges. To gain root privileges you
have to use an exploit. We could not find a rooting method that prevented any
changes to the phone’s file system (the exploit has to be loaded into a partition
on the phone), but we settled on the small ”rageagainstthecage”-exploit, which
is an ADB setuid exhaustion attack. In a nutshell, the exploit crashes the ADB
daemon (adbd) on the phone and prevents it from deescalating root privileges
when it restarts. We know of no trusted exploits for forensics, and this seemed
the soundest approach to minimize changes to the running operating system on
the device (16). After uploading the exploit to the phone and executing it, the
shell should have temporary root privileges till the next reboot of the system.

As seen in Listing 2.3, we can now set /data/misc with sufficient permissions to
write and read the heap-dump we want to generate, because it will automatically
be dumped in that directory.

```bash
# chmod 777 /data/misc
chmod 777 /data/misc
```

Listing 2.3: Setting the needed permissions for /data/misc

Hereafter we send the SIGUSR1-signal\(^3\) to the process. The most common way to
do that, is sending the kill-command and ”-10” as its parameters to the process
represented by its PID. In our example we will kill the ”com.android-browser”-
process with its PID 180, as shown in Listing 2.4.

```bash
# kill -10 180
kill -10 180
```

Listing 2.4: Killing the process with the kill-command

\(^3\)In the davlik vm the SIGUSR1-signal is defined to force the Garbage Collector and hprof
heap dumping
2.2. Acquisition of Heap-Dumps till Android 2.2

By using the "logcat"-command in the shell, we can check the log-file if the dump is generated correctly as Listing 2.5 shows.

```
I/dalvikvm( 180): threadid=3: reacting to signal 10
I/dalvikvm( 180): SIGUSR1 forcing GC and HPROF dump
I/dalvikvm( 180): hprof: dumping VM heap to "/data/misc/heap−dump−tm1335203557−pid180.hprof−hptemp".
I/dalvikvm( 180): hprof: dumping heap strings to "/data/misc/heap−dump−tm1335203557−pid180.hprof".
D/dalvikvm( 180): GC,HPROF_DUMP_HEAP freed 324 objects / 55096 bytes in 3087ms
```

---

Listing 2.5: Checking if the dump is generated correctly via logcat

If Android 2.3 or higher is used, logcat would produce the output displayed in Listing 2.6, when we killed the process with the PID 363.

```
I/dalvikvm( 363): threadid=4: reacting to signal 10
I/dalvikvm( 363): SIGUSR1 forcing GC (no HPROF)
D/dalvikvm( 363): GC,EXPLICIT freed 179K, 58% free
2886K/6727K, external 2118K/2645K, paused 52ms
```

---

Listing 2.6: Checking if no hprof-file is generated via logcat
As shown above, the SIGUSR1-signal forces the GC but no hprof-file is generated (17).

When we change into the /data/misc-directory, we see the generated heap-dump as a "hprof"-file as shown in Listing 2.7.

```bash
# cd /data/misc
cd /data/misc
# ls -l
ls -l
-rw-rw-rw- app_11 app_11 3110245 2012-03-20 23:33 heap-dump-tm133
2286423-pid180.hprof
drwxrwx--- bluetooth bluetooth 2012-03-20 23:12 hcid
drwxrwx--- keystore keystore 2012-03-20 23:12 keystore
drwxrwx--- system system 2012-03-20 23:12 vpn
drwxrwx--- wifi wifi 2012-03-20 23:12 wifi
```

Listing 2.7: Listing /data/misc with ls -l
In the following step we will transfer the dump-files to our local machine as you can see in Listing 2.8. The ADB provides us with a command named "pull". By using it, the files will be downloaded to our machine in the chosen directory. Before that we have to exit the shell with the "exit"-command.

D:\AndroidSDK\tools>adb pull
/data/misc/heap--dump--tm1332286423--pid180.hprof.
1109 KB/s (3656449 bytes in 3.217s)

Listing 2.8: Pulling the dump-file to the local machine

If you use an older Android version than 1.5, there will be two generated output files named xxx.hprof and xxx.hprof-head, where "xxx" is a runtime-generated value that ensures the filename is unique. We have to transfer both dump-files to our local machine. After the transfer we have to merge these two files by using either the "cat"-command on a linux-system or the "type"-command on a windows-system as you can see in Listing 2.9

Linux:

```
# cat xxx.hprof--head xxx.hprof > dump.hprof
```

Windows:

```
type xxx.hprof--head > dump.hprof
type xxx.hprof >> dump.hprof
```

Listing 2.9: Merging the generated dump-parts
2.3. Acquisition of Heap-Dumps since Android 2.3

In this section we will explain how to acquire a heap-dump for Android 2.3 till Android 4.0 by using the DDMS-Tool (2) that is located in the tools-directory of the AndroidSDK. We already depicted that on Android, every application runs in its own process, each of which runs in its own virtual machine (VM). Each VM exposes a unique port that a debugger can attach to.

When DDMS starts, it connects to the ADB. When a device is connected, a VM monitoring service is created between ADB and DDMS, which notifies DDMS when a VM on the device is started or terminated. Once a VM is running, DDMS retrieves the VM’s process ID via ADB, and opens a connection to the VM’s debugger through the ADB daemon (adb) on the device. DDMS can now talk to the VM using a custom wire protocol.

DDMS assigns a debugging port to each VM on the device. Typically, DDMS assigns port 8600 for the first debuggable VM, the next on 8601, and so on. When a debugger connects to one of these ports, all traffic is forwarded to the debugger from the associated VM. You can only attach a single debugger to a single port, but DDMS can handle multiple, attached debuggers.
By default, DDMS also listens on another debugging port, the DDMS "base port" (8700, by default). The base port is a port forwarder, which can accept VM traffic from any debugging port and forward it to the debugger on port 8700. This allows you to attach one debugger to port 8700, and debug all the VMs on a device. The traffic that is forwarded is determined by the currently selected process in the DDMS Devices view. The following figure shows the button for generating a heap-dump.

Figure 2.2.: DDMS-screenshot for hprof-dumping
2.4. Heap-Dump Conversion

The generated heap-dump contains specific information related with the Dalvik VM, like whether something was allocated in the "main"- or "zygote"-heap (8) and that is not part of the standard J2SE hprof format, so the profiling tools like MAT or Visualvm won’t be able to open it. Android at its core has a process they call the "Zygote", which starts up at init. It gets its name from the dictionary definition: "It is the initial cell formed when a new organism is produced". This process is a "warmed-up" process, which means it is a process that has been initialized and has all the core libraries linked in. When you start an application, the Zygote is forked, so now there are 2 VMs. Now a real speedup is achieved by not copying the shared libraries. This memory will only be copied if the new process tries to modify it. This means that all of the core libraries can exist in a single place because they are read only.

Luckily, the Android team has released a tool named "hprof-conv" to trim the Dalvik specific records and convert the dump to a standard format. The figure below shows how to use it.

Listing 2.10: Converting the dump

D:\AndroidSDK\tools\hprof-conv dump.hprof standard --dump.hprof
2.5. Heap-Dump Acquisition Python-Script

The Python-script (20) presented in Listing 2.11 will automate the whole process explained in Section 2.2 and 2.4. It is tested on Ubuntu Linux 11.04 x64. In order for the script to work correctly, the directories where the ADB and the hprof-conv-tool are located need to be part of the PATH environment variables and furthermore no hprof-files and hprof-head-files should be present in the directory of the script. The script mainly uses the subprocess-module of Python, which allows the user to spawn new processes. By using this module we can execute the ADB with the needed parameters. However, in some cases we have to wait for the ADB to terminate, so we suspend the execution of the whole script by using the ”time.sleep”-method for the time needed by the ADB to terminate. First of all we list the processes of the phone we are connected to with the ps-command. The user now has to select the process he wants to kill by entering its PID. After that the script executes the remaining commands to acquire the dump automatically. After the execution you will find a ”dump.hprof”-file in the directory of the script, that is working fine with MAT.

```python
@authors: simon

#!/usr/bin/python2.6
import shlex, time, os, subprocess as sub

p = sub.Popen(['adb', 'shell', 'ps'])
SECONDS_TO_WAIT = 1
time.sleep(SECONDS_TO_WAIT)
str = raw_input("Enter your input: ")
print "Received input is : ", str
args = str.split()

# Pround and kill the process

d = sub.Popen(['adb', 'shell', 'chmod 777 /data/misc'])
h = sub.Popen(['adb', 'shell', 'kill -10', str])
SECONDS_TO_WAIT = 10

# Change to /data/misc

time.sleep(SECONDS_TO_WAIT)
k = sub.Popen(['adb', 'shell', 'cd', '/data/misc'])
l = sub.Popen(['adb', 'shell', 'ls', '\-l', '/data/misc'])
```

sub.Popen(['adb', 'shell', 'exit'])
sub.Popen(['adb', 'pull', '/data/misc/'])
SECONDS_TO_WAIT = 17
time.sleep(SECONDS_TO_WAIT)
sub.Popen(['adb', 'shell', 'rm', '/data/misc/* .hprof'])
for filename in os.listdir("."):  
  if filename.endswith(".hprof−head"):  
    test = "tester.hprof−head"  
    os.rename(filename, test)
for filename in os.listdir("."):  
  if filename.endswith(".hprof"):  
    test2 = "test.hprof"  
    os.rename(filename, test2)
os.system("cat tester.hprof−head test.hprof > temp.hprof")
for filename in os.listdir("."):  
  if filename.startswith("temp"):  
    temp= "tester.hprof"  
    os.rename(filename, temp)
else:  
  temp = test2
newname = "dump.hprof"
sub.Popen(['hprof−conv', temp, newname])
print (l)

Listing 2.11: Source: dumpacquisition.py
3. Analysis of the acquired Heap-Dumps

After showing how to acquire heap-dumps for certain processes of a running process we analyze in this section the most widely used Android apps like the Facebook-App, the Twitter-App or the GoogleMail-App in a forensic manner by using the heap profiling tool MAT. The Facebook-App is one of the most downloaded Android-Apps, between 100.000.000 and 500.000.000 downloads since its launch. The Twitter-App gets more and more popular these days with social networking and web2.0-applications are getting more important in our every day life. So its download figure aggregates between 50.000.000 and 100.000.000. The GoogleMail-App is installed on nearly every mobile phone running Android (10).

Furthermore we will differ between the ”logged in”- and the ”logged out”-state of these apps, so we can see which data is still present in the memory after logging out of an app what could be a even bigger security hole.
3.1. Explanation of the MAT-Tool

The Eclipse Memory Analyzer is a fast and feature-rich Java heap analyzer that helps finding memory leaks and reducing memory consumption. In our case, we will use MAT mainly for searching strings that are present in the generated heap-dumps. One useful feature of MAT is the search-engine that uses the Open Query Language (OQL) (13). OQL is a query language standard for object-oriented databases modeled after SQL. If you know the exact string you want to search for, you can find it very easily with that feature. In the example below, we search for the string "xxx".

```
SELECT * FROM java.lang.String s WHERE toString(s) = "xxx"
```

Listing 3.1: OQL-Search for the string "xxx"

Although you could find a known string very easily with that feature, a forensic investigator normally does not know a username or password of an account he investigates. Therefore our approach looks like this:

- Starting MAT and loading the hprof-file
- Opening the dominator tree
- Searching for "class java.lang.String"
- Listing all objects with incoming reference of the System class "java.lang.String"
- Expanding the whole list and saving it to a txt-file.

The Facebook-Dump for example contains nearly 18.000 string-entries, so it can take some time to expand all of them depending on your machine. Saving it into a txt-file will provide us with the opportunity to search through it in the common windows-editor or extracting data with scripts that are using regular expressions (11).
3.2. Memory Analysis of the most widely used Android-Apps

3.2.1. Facebook-App

The first app we will investigate, is the Facebook-App. As we depicted, it is the most downloaded app of the Android-Market and widely used. For that reason it is very interesting for a forensic investigator to get information of a Facebook-account. We use the app-version 1.8.4 with the sha256-hash "92f5be800bae1a96b4dad647a3aa7e84a8694f4a74c4f5984d0dd0c73970603" and we will differ between the "logged in"- and "logged out"-state of the app.

"Logged in"-state

In this state we can find a great deal of interesting information, even the username and password is visible in the plaintext.

Searching for the password of your own Facebook-account in the generated txt-file is not very difficult. However, finding a unknown password in a heap-dump is increasingly more difficult. After analyzing multiple Facebook-dumps, we noticed that the string "pwd" can be easily connected to the password-string because every time we analyzed a Facebook-dump, we could find the pwd-string one line after the password-string, in this case "abc123" as you can see in the Listing below.

|− <class> java.lang.String @ 0x405b58a8 GTE CyberTrust Global Root
|− <class> java.lang.String @ 0x405b5718 abc123
|− <class> java.lang.String @ 0x405b56d8 pwd
|− <class> java.lang.String @ 0x405b5670 2
|− <class> java.lang.String @ 0x405b5630 sid

Listing 3.2: Search for the string "pwd" in a dump of the Facebook-App
After searching for the password we will search for the username. First of all you have to know that Facebook generally uses an email-address as an username. But there will be more email-address-strings that you can find, for example the email-address that is listed in the profile. Luckily most of the Facebook-App-users use the same email-address for the login like for their profile. The easiest way to extract email-addresses from a txt-file is to use regular expressions (11), sometimes shortened to regex. A regular expression is a special text string for describing a search pattern. In our case we use the regex shown in Listing 3.3.

```
[\w\-]+@[\w\-]+\.[\w\-]+[a-zA-Z]{1,4}
```

Listing 3.3: Email-regex to extract all email-addresses from a txt-file
3.2. Memory Analysis of the most widely used Android-Apps

The Python-script (20) presented in Listing 3.4 displays all email-addresses that are present in the input-file. In the app-version that we use, the second email-address is the one listed in the profile and the third email-address is the one that is used as an username.

```python
import re
foundemail = []

mailsrch = re.compile(r'[^\w\-][\w\-]+@[\w\-]+\@[a-zA-Z]{1,4}')

str = raw_input("Enter the path to your txt-file: ")
for line in open(str, 'r'):
    foundemail.extend(mailsrch.findall(line))

print foundemail
```

Listing 3.4: Source: emailaddress.py

Not only the email- and password-information is visible, you can also find more useful information of a profile like the employer, the birth date or hometown if the user specified that in his profile. Messages or bulletin board entries are not visible in the heap-dump.

"Logged out"-state

In this state the password is not visible anymore but you can still find the username with the Python-script displayed in Listing 3.4. The other profile-information is still visible in the heap-dump!
3.2. Memory Analysis of the most widely used Android-Apps

3.2.2. Twitter-App

Another widely used app is the Twitter-App. We use the app-version 3.0.1 with the sha256-hash
"e3b0c44298fc1c149aafbf4c8996fb92427ae41e4649b934ca495991b7852b855" and we will differ between the "logged in"- and "logged out"-state of the app.

"Logged in"-state

Finding the username and password of the used Twitter-account was our primarily goal but after analyzing the dump we came to the conclusion that only the username is present in plaintext. By searching for the string "account_name" you will get many results, but in one case you get the output shown in Listing 3.5. When "account_name" is referenced with something other than the string "unknown" you have found the username. In our example we used the username "testuser". After finding the username you will realize that it is present very often in the dump but the explained way of finding it, is the best and most easy way.

```
|− <class> java.lang.String @ 0x4085ce38 id
|− <class> java.lang.String @ 0x4085c9e8 SELECT id, tweet, mention, message, discover FROM activity_states WHERE (account_name='testuser')
|− <class> java.lang.String @ 0x4085c598 testuser
```

Listing 3.5: Search for the username in the Twitter-App
3.2. Memory Analysis of the most widely used Android-Apps

Not only is the username present in the dump, other useful information can be found too. For example all the Tweets of the user and the name of the Twitter-user as you can see in Listing 3.6. In the first line you see the name of the Twitter-user, in the second line the Tweet and in the third line the username.

| – `<class>` java.lang.String @ 0x4078d980 Tester Testing |
| – `<class>` java.lang.String @ 0x4078d940 test−entry |
| – `<class>` java.lang.String @ 0x4078d8f0 testuser |

Listing 3.6: Search for a Tweet in the Twitter-App

As we depicted the password is not present in plaintext in the Twitter-App unlike the password of the Facebook-App. The reason for that is that the Twitter-App uses the OAuth 1.0 Protocoll (18) as you can see in Listing 3.7. OAuth is an authentication protocol that allows users to approve application to act on their behalf without sharing their password.

| – `<class>` java.lang.String @ 0x407410e0 http://api.twitter.com/ |
| – `<class>` java.lang.String @ 0x40740f48 OAuth realm=%s”, oauth_version="%s”, |
| oauth_nonce="%s”, oauth_timestamp="%s”, oauth_signature="%s”, |
| oauth_consumer_key="%s”, oauth_signature_method="%s” Unknown |

Listing 3.7: Search for the OAuth-token in the Twitter-App

In the traditional client-server authentication model, the client uses its credentials to access its resources hosted by the server. As a result of the increasing use of distributed web services and cloud computing, third-party applications require access to these server-hosted resources. OAuth introduces a third role to the traditional client-server authentication model: the resource owner. In the OAuth model, the client (which is not the resource owner, but is acting on its behalf) requests access to resources controlled by the resource owner, but hosted by the server. In addition, OAuth allows the server to verify not only the resource owner authorization, but also the identity of the client making the request.
OAuth provides a method for clients to access server resources on behalf of a resource owner (such as a different client or an end-user). It also provides a process for end-users to authorize third-party access to their server resources without sharing their credentials (typically, a username and password pair), using user-agent redirections.

For example, a web user (resource owner) can grant a printing service (client) access to her private photos stored at a photo sharing service (server), without sharing her username and password with the printing service. Instead, she authenticates directly with the photo sharing service which issues the printing service delegation-specific credentials.

"Logged out"-state

In this state nothing changes concerning the information that is present in the dump. The username, the name of the Twitter-user and all Tweets are still present in plaintext and detectable in the same way as in the "logged in"-state.
3.2. Memory Analysis of the most widely used Android-Apps

3.2.3. GoogleMail-App

On the majority smartphones using Android have Google-Apps pre-installed. So the GoogleMail-App is used by nearly every Android-user. Android and all Google-Apps are connected in a special way. You can not logout of any of these apps like you can do in your Facebook- or Twitter-App. If you want to be logged out in your GoogleMail-App, you have to kill the whole process. Therefore we don’t differ between the "logged in"- and "logged out"-state in this section because most users do not do that in order to logout of the GoogleMail-App. We use the app-version 2.3.5 with the sha256-hash "4877528e25138b8771a9d8473fc901941c864308a238b9cb6b7da645ac10e06a”.

By analyzing the GoogleMail-App our main goal is finding the username and password of the used GoogleMail-account. But after analyzing the dump we came to the conclusion that only the username is present in plaintext. The username is identical to your GoogleMail-address. In order to find it in the dump, you have to search for the string "googlemail" as you can see in Listing 3.8 where we found our test-username "testaccount123@googlemail.com".

```
|− <class> java.lang.String @ 0x40611e48 −3
|− <class> java.lang.String @ 0x406107e0 testaccount123@googlemail.com
|− <class> java.lang.String @ 0x40610790 authAccount
```

Listing 3.8: Search for the username in the GoogleMail-App

The Python-script displayed in Listing 3.4 can be used to find the username too.
Further to the username, the in-box of the account is visible in the dump. Finding that information is very easy. If you open your generated txt-file, some of the first lines represent your in-box as you can see in Listing 3.9. Unfortunately the whole Mail couldn’t be found in the dump.

Like the Twitter-App, the GoogleMail-App uses the OAuth 1.0 Protocoll as its authentication protocol. So there is no password visible in plaintext in the dump.
4. Conclusion and Further Work

The presented methods of acquiring and analyzing heap-dumps of smartphones running Android will be summarized in Section 4.1. After that we explain in Section 4.2 some limitations of the presented methods. The next Section 4.3 introduce some future work that could be done. In Section 4.4 we explain our final conclusion.

4.1. Summary

In this Student Research Paper, we identified the need for a live volatile memory forensic analysis for smartphones. We described some basics about the Dalvik VM memory usage. We have explained the way of acquiring heap-dumps of a process using the ADB or the DDMS-tool. In addition to that we showed why we need to convert the acquired heap-dumps. Furthermore we automated the acquisition of these heap-dumps for smartphones running Android till version 2.2 within a Python-script.

Moreover we analyzed three of the most widely used Android-Apps with a heap-dump analyzing tool called MAT, concentrating on finding usernames, passwords and other information that could be useful for forensic investigators.

4.2. Limitations

As we depicted in Section 1.2 the python-script that we presented in Section 2.5 is not working for the newer Android versions starting with version 2.3. An automated way of acquiring heap-dumps for these newer versions of android is not part of this paper.

Aside from that David Poll described a way how malicious Android apps impersonate you on your Facebook account in April 2012 but he discovered that bug
already in February 2012 and informed Facebook about it (19). Facebook asked Poll to handle that topic with discretion so that nobody could use that bug and that they have enough time to close that security-hole. Facebook released an update for the Facebook-App by the end of March 2012 where they eliminated that bug. In addition to that passwords are no longer visible anymore in plaintext in the heap-dump. Therefore the results of our analysis of the Facebook-App can only applied to the Facebook-App version that we use in this paper.

4.3. Future Work

In future work there should be put some emphasis on automating the acquisition of heap-dumps for smartphones running Android 2.3 or higher within a Python-script. Furthermore the information that we found in the heap-dumps that we presented in Chapter 3 should be saved securely in the memory of the smartphone and not in plaintext.

4.4. Conclusion

The increasing use of apps on smartphones opened a profit-yielding field for forensic investigators. As we proofed, a lot of interesting information that could be relevant in criminal investigations are available by dumping a process memory to your phone. Regarding the results of the analysis of the Facebook-App and finding the password of the used Facebook-account in plaintext, we have not expected to find such a big security hole in a software that is used by so many people around the world and is produced by one of the biggest and state-of-the-art companies concerning web2.0-technologies. Facebook may have closed that security hole but there are still many people out there who do not update their apps regularly.
Bibliography


17 McFadden, Andy: Don’t do heap dump on SIGUSR1, Accessed:04/19/2012. [Online]: https://github.com/android/platform_dalvik/commit/b037a464512c0721bdca969ae19c3d4b17b083#vm/SignalCatcher.c.


Appendix
A. How to get temporary root privileges

If you want to know how to get temporary root privileges on a smartphone running Android, we will explain in the following Sections how to do that by using one common exploit.

Source: http://forum.sdx-developers.com/?PHPSESSID=0m1ibtahmt818e12nkvirs1f7&topic=14067.0

We assume that the ADB and the USB drivers are installed correctly on your workstation. Furthermore the ADB should be part of your PATH environment.

A.1. Download and Extract the rageagainstthecage-arm5.bin-exploit

You have to download the rageagainstthecage-arm5.bin-exploit from the internet. Via google you can find many download mirrors. Thereafter you have to extract the bin-file with 7Zip or any other program that can extract zip-files.

A.2. Activate USB-Debugging on your target phone

To activate USB-Debugging you have to follow these steps:

- Press Home Key
- Press Menu Key
- Select Settings
- Select Applications
- Select Development
- Check USB debugging on
A.3. Start the ADB-Server and copy the exploit to the phone

- Check Stay awake on

After that you have to connect your phone via USB to your workstation.

A.3. Start the ADB-Server and copy the exploit to the phone

You have to open a command prompt window and type the following commands:

- `adb start-server`
- `adb devices`
- `adb push rageagainstthecage-arm5.bin /data/local/tmp/rageagainstthecage-arm5.bin`

A.4. Run the exploit on your phone

Open a shell on your phone via ADB, set the exploit with sufficient permissions and run it:

- `adb shell`
- `chmod 755 /data/local/tmp/rageagainstthecage-arm5.bin`
- `/data/local/tmp/rageagainstthecage-arm5.bin`

A.5. Check whether you have gained root privileges

Open a shell on your phone via ADB. If you see # prompt instead of $ prompt, you have gained root privileges till the next reboot of your system.
Eidesstattliche Erklärung


Ich bin ferner damit einverstanden, dass meine Arbeit zum Zwecke eines Plagiatsabgleichs in elektronischer Form anonymisiert versendet und gespeichert werden kann. Mir ist bekannt, dass von der Korrektur der Arbeit abgesehen werden kann, wenn die Erklärung nicht erteilt wird.

Nuernberg, den 10.05.2012

Simon Leppert