Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 00 (2014) 1–11

Procedia Computer Science

Android mempry forensics

Abstract

Smart phones today are getting ever more powerful, and capable of doing most of the tasks usually associated with computers. This have made them increasingly interesting in forensics investigations and information theft. In this paper we present the findings we have done when looking at the volatile memory of a virtual Android device. In our research we have used string search, hex editor and other tools to search through the memory to find cleartext information to prove what sort of information that are stored here. We also discuss the possibility and usefulness of automated forensics tools to look through smart phone memory and the difficulties faced when developing such tools. Another aspect of our research was to look at secure applications to see whether or not the volatile memory is taken into account in development. This research provides a baseline for further development of memory analysis tools, proving its usefulness and what information you can expect to find on the subject.

Keywords:

1. Introduction

In digital forensics, memory is an important source of evidence. Most of the data from applications is at some time stored in the memory of the device. Modern smartphones use memory of larger sizes such as a conventional computer (laptop, desktop etc.). The memory in these devices contain user and application data which might be valuable in forensic investigations. Data kept in the memory include information about running and terminated processes, open files and network information [1]. Other valuable information stored in memory can be image files, GPS data, web-history and encryption keys. Encryption keys might be highly valuable as forensic evidence. If the encryption keys are acquired, a investigator could get access to valuable information that is encrypted.

There are several ways of acquiring memory from an Android device. When having hardware access its possible to extract memory from the chips on the device, other methods by using software would in most cases require a unlocked bootloader and root access. As presented by [1], one way to get root access is to use a privilege escalation exploit.

Memory is highly volatile, acquiring it can therefore be a challenge. Memory is considered volatile for multiple reasons. It requires power to be kept present [**?**] and gets written to approximately every $10ns$ in regards of order of volatility. It is not possible to use a write-blocker on memory as you could when acquiring evidence from for instance a hard drive or a flash memory medium. Therefore, to the same degree as when dealing with other sources of digital evidence, it is important to gather the evidence as soon as possible.

Different type of applications use different amounts of memory. Games use more memory than applications that is not as graphical. Since games allocates more memory, also dynamically, objects not being used in the memory can

be erased. This can result in loss of evidence that may be useful in an investigation.

Address Space Layout Randomization (ASLR) makes analysing the memory of processes more difficult, because the memory is not laid out in a linear fashion. This means that you have to leverage global data structures more for locating where in memory data will be. ASLR will also allocate the data in different parts of the memory each time the same process is started. This means that some piece of data, belonging to one process, will not be in the same place in memory every time you start the process. ASLR was mainly developed for preventing buffer overflows [**?**]. But it also makes it more difficult to analysing memory.

2. Related Work

In the last years there has been done a lot of research on acquisition of memory and analysis techniques, targeting Linux and Android. The most common way to gain access to memory is gaining root privileges and load a new kernel into the phone. While this is not ideal, as it's overwriting some of the memory in the progress, it is the only known way to get access to memory of all running processes.

In 2011 J. Sylve et al. presented a paper that described a forensic sound approach of acquiring Android memory. [1] This paper looked into ways of obtaining memory, with different tools and different approaches. In their experiment, with an emulator, they performed a memory dumping method that is called *pmemsave* and compared it to other ways of dumping memory. When compared to another method called Droid Memory Dumpstr (DMD) it was over 99% identical. Since the memory dumps takes time, in our experiment it took us 5 minutes to dump 800mb of memory, some changes will naturally occur in the memory.

Based on these results, they believe that this method for dumping memory is a forensic sound process and can be used as evidence in court. To analysis the data extracted, different methods can be used, in this project they also used Volatility. One of the key benefits of this open source program is the ability to make your own plugin. They made a plugin that separates the memory that belongs to a single process to make it much easier to manually analyse it. Unfortunately we were not able to find this plugin to test it.

This may be the first paper published that presents a method of accurate memory acquisition on Android, as they write in their conclusions "To our knowledge, this is the first published work on accurate physical memory acquisition and deep memory analysis of the Android kernel's structures".

3. Method

3.1. Methodology and approach

3.1.1. Alternatives

We found several alternatives for acquisition of memory from Android devices.

dd on `/dev/mem`

This is a very simple method for acquiring memory. But `/dev/mem` can only be used directly when the kernel is compiled with the `STRICT_DEVMEM` flag off or with a kernel version pre 2.6. The first kernel version ever used on Android is version 2.6.25[1]. There is also a problem with the dd application in Android devices, it can not handle file offsets above `0x80000000`[1].

fmem

fmem is a LKM (Loadable Kernel Module) which creates `/dev/fmem`. `/dev/fmem` is similar to `/dev/mem`, but without the limitations. However, there are some issues using fmem on Android devices running ARM including the problem with `dd` mentioned earlier [1].

[1] `http://elinux.org/Android_Kernel_Versions`

LiME LKM

LiME (Linux Memory Extractor) LKM[2] provides a forensically sound method for acquiring memory from memory [2]. LiME is formerly known as DMD (Droid Memory Dumpstr).

3.1.2. Chosen approach

For acquisition of memory we have chosen to use the LiME LKM because of the problems described with the other methods.

3.2. Environment

When starting the project we needed to decide weather to use an physical device or an emulatorto conduct our experiment on. To make the experiment as close to reality as possible it should have been done on a physical device, however, an emulator gives us a close match to reality and is easily replicable. Also, by using an emulator we avoid rooting our own phones, thus we avoid voiding warranty. Also, we found a guide on the Volatility wiki [3] for how to dump memory with an Android emulator. Therefore we chose to use an emulator. Some deviation from the guide was done to be able conduct the experiment, due to the fact that we were running on a Linux system, not Mac OS X as the guide uses. This mostly involves editing the paths in the Makefiles you are editing to fit your system. A detailed guide on the setup can be found in the appendix.

The emulator was set up on a computer running the latest Ubuntu (Ubuntu 14.04.1 LTS x86_64). The emulator we set up is based on the Nexus 7 (2012) with Android 4.2.2 Jelly Bean (API level 17) and The Linux Kernel 2.6.29. This was chosen because of convenience since the guide we followed is using this and the 2.6 Kernel have very few restrictions.

3.3. Tools

3.3.1. Volatility

Volatility is a open source collection of tools used to extract digital artefacts from memory.[4] The framework is not meant for a live forensic analysis but an offline analysis on memory acquired from a live machine or device, see Section 3.1.1 for memory acquisition. Volatility has support for Windows, Mac OS, and Linux, and from version 2.3 they also supported the ARM architecture, this is essential for analysing memory on Android devices, as most devices that run Android has a ARM architecture. With a memory dump, Volatility can extract information such as running processes, network connections, mounted devices and view the mapped memory for processes and kernel modules. Volatility is especially useful when analysing a system compromised by malware, as malware often hides itself in memory to avoid detection. With Volatility the artefacts of the malware can be found, and be analysed.

Volatility can also help when encryption is used on the device, if the device is powered down the keys will be lost and you might be unable to use the data collected from a less volatile source such as a SD card or a hard drive. With Volatility the keys can be extracted from the memory (if present), Volatility comes with a plugin that can be used to extract the key used by Truecrypt for disk encryption. In our case we wanted to see if we could find the encryption key used by a popular SMS app, TextSecure, to encrypt messages.

The easiest approach to the problem of finding data used by Android applications is to go through the Dalvik VM as proposed by Case[3]. The Dalvik VM is the *process virtual machine*[5] for Android that is used to run Android application, applications are translated to Dalvik bytecode. This makes the Dalvik VM a good entry point for analysing applications. The first application that starts and runs in a Dalvik VM is the *zygote* process. It preloades and initializes classes that are shared between different processes, it is also the process that will fork additional Dalvik VMs if required. The Dalvik VM is implemented as a shared library which is loaded dynamically by applications.

[2] https://github.com/504ensicsLabs/LiME
[3] https://github.com/volatilityfoundation/volatility/wiki/Android
[4] https://github.com/volatilityfoundation/volatility
[5] https://en.wikipedia.org/wiki/Dalvik_%28software%29

This can be seen by the `pstree` and `proc_maps` plugins, see Appendix **??** and **??**.

Since all the user applications on Android runs in a Dalvik VM it makes a good entry point for forensic analysis, the source code is also publicly available. It is written in C++, but makes use of simple C structures, which makes it easier to locate them in memory compared to more complex structures such as classes. Case and [4] shows that the *DvmGlobals* structure which is declared in *dalvik/vm/Globals.h* contains a lot of information about the application it is running. The *loadedClasses* member is a pointer to a hash table which contains all the system classes, this makes it a valuable structure because it can point us to where in memory information can be found.

Volatility has a plugin interface for creating plugins. It is possible to invoke other plugins from your own, this means you can stitch together several other plugins to write your own. This makes it much easier to write plugins, since you do not have to reinvent the wheel for every plugin. There are plugins for analysing the Dalvik VM, but they are very version specific.

3.3.2. PhotoRec

File recovery tool, supports many platforms and file systems, including memory images. It Can extract images, text files and more.

3.3.3. Strings

Easy tool for getting all strings in a dump.

3.3.4. Hex editor

Useful for looking trough a dump.

4. Experiments

We did five dumps of the memory;

4.1. Clean dump

The first dump we did was right after boot, with factory settings and nothing done to the device other than transferring the LiME LKM to the SD-card.

The reason for this dump so we could see if we had set up our environment correctly and could read memory of the phone. The dump was also great for use when comparing to later dumps where we could see differences from a system with no significant use and compare them to the other experiments.

Strings.

Volatility. With volatility and its plugins we where able to see the process list [6] and other use full information about the system.

PhotoRec. This program managed to get alot of files, such as java code files and txt files containing information from internal logfiles as syslog (dmesg). It also recovered some png files with pictures used by the Android launcher.

4.2. Pastebin entry

The second dump we did was after creating a pastebin entry on pastebin.com using the stock Android browser (for 4.2.2), to see if we could find our entry in the memory.

When starting to analyze this dump, we first started with the most basic. Strings and hex editor; by searching for ”pastebin.com”. Other findings in this process included a ”timeline” on how the user got to the page patebin.com by examining the memory segments before the hit on our string.

[6] `https://github.com/volatilityfoundation/volatility/wiki/Linux%20Command%20Reference#linux_pslist`

Hex editor. This gave us a good way to examine the dump at its lowest level, by searching for ”p.a.s.t.e.b.i.n...c.o.m” we found the url we posted in our emulator

Strings. Strings would show alot of results since there are often many hits on readable data in these kind of dumps, by piping the output to grep we where able to filter out text we wanted. Simply searching for pastebin.com we where able to find the page it was posted on.

4.3. Standard Text Message

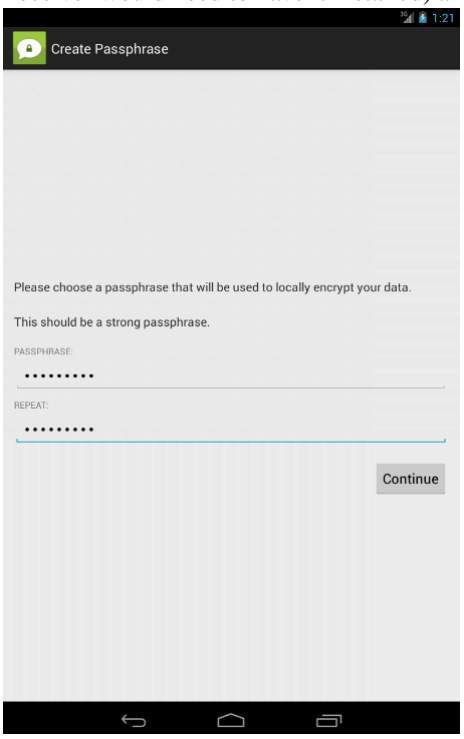
The next dump was after sending a text message to the standard messages application. Since the emulator provides an telnet interface, we where able to send a message using `sms send <phoneNumber> <textMessage>`.

This experiment was to see if it was possible to get recently received text messages from a live system for use in a forensic environment.

Strings. Same as above

4.4. Secure Text Message

Next we did the same as above, but after installing TextSecure[7], which encrypts the messages in a database. TextSecure was chosen since its a commonly used application for securing your text messages in transit (both sender and receiver would need to have it installed) and open source so we could get a better understanding on how it works.



In some cases the device might have used anti-forensic tools to hide their activity, we wanted to look into what information a memory analysis could retrieve.

4.5. Screen lock

We also did a memory dump after creating a screen lock with a PIN code and using a passphrase.

If the device has a unlocked bootloader it would often be possible to use out method to retrieve memory of a device without wiping off all data from the device. This experiment was done to see if we could find a passphrase or incode from the memory dump.

[7] `https://play.google.com/store/apps/details?id=org.thoughtcrime.securesms`

Strings. By searching for the pincode and passphrase we saw a pattern

```
tomme@tomme-VirtualBox:~$ strings ~/lime-dumps/lime10.dump | grep jallapass
jallapass
jallapass6dc65f8042ac1724
jallapassVPN
jallapass6dc65f8042ac1724I
```

5. Discussion

In our project we where forced to discuss some issues for the project to be possible in our time frame. Our choice would need to be in a forensic sound manner so they could be applicable in a court.

5.1. Real phone vs. emulator

In a real case scenario, a forensic investigator would receive a physical device that the investigator should acquire evidence from. However in this project there are a number of reasons for why we did not use a physical device.

While many companies give instructions on how to unlock their bootloader, they also state that the warranty is voided. However law-regulations in many European countries have better consumer laws that give the consumer rights if the fault is not related to ROM or kernel of the device. Although we therefore would not have any real risk of loosing our rights to the producer we choose not to use our own devices since there always is some risk involved where the device could be bricked and consumer laws would not be applicable.

When conducting scientific experiments, it is important to do something that is reproducible. By using a emulator we can document how it is set up so others can copy the environment and get the same results. Physical devices could be a issue where the device is out of production.

5.2. Secure application - TextSecure

One of our experiments was to look into how an application that gives the user higher privacy by encrypting its messages. TextSecure also enables encrypted point-to- point encryption by use of PKI. As our results have shown we did not see the message in the memory dump such as we did when tested against the stock SMS application. However, it might be possible to extract the encryption keys where the database was acquired and decrypt the messages. Since TextSecure is open source it is possible to look through it's code and see how it works. We know that at some point they have used a memory cleaner to clean sensitive memory after it has been used. [8] The memory cleaner class was in use in the version of TextSecure we tested, so this may be why we did not find the messages in memory. We are unsure if the key is available in memory, as a key derivation algorithm is in use to generate a more suitable key from the user made password. For this last problem Volatility could have been used if we had a working version of the Dalvik plugins in time. Then we could have made a plugin that leverages the plugin that finds the loaded classes, and found where in the relevant TextSecure class the keys are stored. This might require that TextSecure is open on the device, or else the memory cleaner might overwrite the keys.

5.3. Issues during the project

Under the project we had some obstacles, we got a good start with researching and reading existing papers on the subject. However after we had decided on what direction we would take the project there where multiple issues when building the prerequisites for the kernel and emulator. There was some confusion from the guide on Volatility page on what to follow. Since the guide mentioned OS X specifically we did not pay close enough attention to what we would have to deviate from it.

5.4. Method

As stated earlier in the report we had a number of methods we could approach obtaining memory from mobile devices. When choosing method we where looking for one that gave us the most forensic sound evidence. Originally our chosen method was using *pmemsave* that was mentioned in a earlier paper[1]. However this did not work in the emulator, so the next best approach was chosen, where research suggests it gives 99.46% identical memory dumps[1].

[8] `http://goo.gl/mgDKp0`

5.5. Applicable in forensic investigation

When doing research on acquiring evidence it is important to see how it correlates to the real world. If the research is highly theoretical and not applicable it could be seen as useless. Here we discuss our thoughts on what would be different from a forensic investigation and our experiments.

5.5.1. Kernel Module

As explained in earlier chapters the LiME kernel module has to be compiled to the running kernel. The kernel also has to support loading of kernel modules. This is by default not allowed since it is a security vulnerability. This might be a problem if the device is running a stock ROM, where it would be necessary to obtain the kernel source from the producer of the device. These are often not released for every device and could be a issue when getting evidence in the field. Since time is a issue when dealing with such volatile evidence it could be feasible to have pre-compiled kernels and modules that would work on most common devices. However this might be known by the criminals.

6. Conclusion

Our project has proved that the volatile memory of Android devices have valuable information for use in a forensic investigation. Each phone has its own ROM and kernel, therefore you would need kernels that support loading kernel modules. These should be pre-compiled since time is a issue when dealing with volatile memory. An alternative would be to purchase hardware toolkits that can extract memory from the chip itself [9]

Encryption and secure applications can make finding forensic investigation more difficult and we have not found any research that has implemented countermeasures against these anti-forensic tools. In newer versions of Android the Dalvik VM has been replaced with Android Runtime(ART) where one of the new features have a smaller memory footprint. This is a issue in the forensic community where there is no plugin available that supports ART.

Our method for analysing the memory is not necessarily realistic when used in forensics investigation, however it proves what might be available in the memory of Android devices. To automate this process might make information from memory easier accessible and more reliable.

Malware could have certain characteristic that could be shown in memory, since mobile phones often stay powered on for longer periods they might be attractive targets for malware only running in memory. This could possibly used for investigating cases with the trojan horse defence.

6.1. Real phone vs. emulator

The major difference between a real phone and a emulator is when acquiring the memory dump is much easier. When using a real phone you run the risk of loosing live memory when trying to cooldboot and memory segments can be overwritten by the time a forensic analyst can take a dump.

The memory dump in it self is representable of a real device and give a clear indication of what information you can expect to find in the memory of Android devices.

7. Further Work

The first and most critical area to make this valuable would be looking into how to actually get the memory of a live device, using the technique we have with the emulator, in most cases require a ”cold” reboot of the device risking degrading the volatile memory.

Another topic for further work would be in developing tools to look through the memory data automatically, as in most cases you do not know exactly what you will be looking for. This would make a tool thats looking for patterns to find information useful. This gets increasingly important as the memory capacity in these devices is increasing quickly. The leap from our emulated device with 800MB memory to newer devices with 2GB and higher capacity would make it not feasible to go through the data manually.

The fact that our research into secure applications is not entirely conclusive, we could not find the encryption

[9] Side med hw tool

key or any messages sent or received using this secure application, but they key is most likely in the memory. How securely hashed or stored this is might be despite the fact it is not in clear text is not currently known. Using a different technique and a deeper search might reveal different results.

Lastly, and this is a topic we intended to research further in our own paper is how long different sorts of information stays in memory. This is probably something that will be hard to test using an emulator, as actual live devices will have a lot more apps and background services running. And you will not be likely to get as clean a memory as we did with our experiments using the emulator.

In this paper we have only looked Android running with the Dalvik VM, in the more recent versions of Android the experimental Android RunTime(ART) virtual machine is introduced. This changes how apps are compiled and run, from Dalviks *just-in-time compilation* to ART's *ahead-of-time compilation*. ART still uses the *dex* bytecode format that Dalvik uses for backward compability, but it is very unlikely that any Dalvik specific Volatility plugins will work with ART. To overcome this plugins needs to be able to read and understand where data is stored in the memory of ART. When this work is done the application specific plugins will only need minor modifications to work with the underlying code that reads the ART instead of Dalvik.

References

[1] J. Sylve, A. Case, L. Marziale, and G. G. Richard, "Acquisition and analysis of volatile memory from android devices," *Digital Investigation*, vol. 8, no. 3-4, pp. 175–184, Feb. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.diin.2011.10.003

[2] A. P. Heriyanto, "Procedures and tools for acquisition and analysis of volatile memory on android smartphones," 2013.

[3] A. Case. (2011) Memory analysis of the dalvik virtual machine.

[4] H. Macht. (2013) Live memory forensics on android with volatility - dipolma thesis. [Online]. Available: http://www.homac.de/publications/Live_Memory_Forensics_on_Android_with_Volatility.pdf

[5] H. Sun, K. Sun, Y. Wang, J. Jing, and S. Jajodia, "TrustDump: Reliable Memory Acquisition on Smartphones," in *Computer Security - ESORICS 2014*, ser. Lecture Notes in Computer Science, M. Kutyłowski and J. Vaidya, Eds. Springer International Publishing, 2014, vol. 8712, pp. 202–218. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11203-9_12

[6] T. Müller and M. Spreitzenbarth, "FROST: Forensic Recovery of Scrambled Telephones," in *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*, ser. ACNS'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 373–388. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38980-1_23

[7] L. K. Yan and H. Yin, "DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis," in *Proceedings of the 21st USENIX Conference on Security Symposium*, ser. Security'12. Berkeley, CA, USA: USENIX Association, 2012, p. 29. [Online]. Available: http://portal.acm.org/citation.cfm?id=2362822

[8] P. Albano, A. Castiglione, G. Cattaneo, and A. De Santis, "A Novel Anti-forensics Technique for the Android OS," in *Broadband and Wireless Computing, Communication and Applications (BWCCA), 2011 International Conference on*. IEEE, Oct. 2011, pp. 380–385. [Online]. Available: http://dx.doi.org/10.1109/bwcca.2011.62

[9] V. L. L. Thing, K.-Y. Ng, and E.-C. Chang, "Live memory forensics of mobile phones," *Digital Investigation*, vol. 7, pp. S74–S82, Aug. 2010. [Online]. Available: http://dx.doi.org/10.1016/j.diin.2010.05.010

[10] T. Vidas, C. Zhang, and N. Christin, "Toward a General Collection Methodology for Android Devices," *Digit. Investig.*, vol. 8, pp. S14–S24, Aug. 2011. [Online]. Available: http://dx.doi.org/10.1016/j.diin.2011.05.003

[11] V. Thing and Z.-L. Chua, "Smartphone Volatile Memory Acquisition for Security Analysis and Forensics Investigation," in *Security and Privacy Protection in Information Processing Systems*, ser. IFIP Advances in Information and Communication Technology, L. Janczewski, H. Wolfe, and S. Shenoi, Eds. Springer Berlin Heidelberg, 2013, vol. 405, pp. 217–230. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-39218-4_17

[12] H. Pieterse and M. Olivier, "Smartphones as Distributed Witnesses for Digital Forensics," in *Advances in Digital Forensics X*, ser. IFIP Advances in Information and Communication Technology, G. Peterson and S. Shenoi, Eds. Springer Berlin Heidelberg, 2014, vol. 433, pp. 237–251. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-44952-3_16

8. Appendix 1

8.1. linux_pstree output

Shortend output from the *linux_pstree* Volatility plugin.

```
tomme@tomme-VirtualBox:~/volatility$ python vol.py --plugins=plugins --pr
ofile=LinuxGolfish-2_6_29ARM -f ~/lime-dumps/lime5.dump linux_pstree
Volatility Foundation Volatility Framework 2.4
Name                 Pid             Uid
init                 1               0
.ueventd             32              0
.servicemanager      33              1000
.vold                34              0
.netd                36              0
.debuggerd           37              0
.rild                38              1001
.surfaceflinger      39              1000
.zygote              40              0
..system_server      292             1000
..GC                 294             1000
..Signal Catcher     297             1000
..JDWP               298             1000
..Compiler           299             1000
..ReferenceQueueD    300             1000
..FinalizerDaemon    301             1000
..FinalizerWatchd    302             1000
..Binder_1           303             1000
..Binder_2           304             1000
..Binder_3           305             1000
..SensorService      306             1000
..er.ServerThread    307             1000
```

8.2. *linux_proc_maps output*

Shortend output from the *linux_proc_maps* Volatility plugin, for the Textsecure plugin.

```
tomme@tomme-VirtualBox:~/volatility$ python vol.py --plugins=plugins --profile=LinuxGolfish-2_6_29ARM -f ~/lime-dumps/lim
e5.dump linux_proc_maps -p 1084
Volatility Foundation Volatility Framework 2.4
Pid      Start              End                Flags       Pgoff Major  Minor  Inode      File Path
-------- ------------------ ------------------ ------ ---------- ------ ------ ---------- ---------
    1084 0x000000002a000000 0x000000002a002000 r-x           0x0     31      0        811 /system/bin/app_process
    1084 0x000000002a002000 0x000000002a003000 r--        0x1000     31      0        811 /system/bin/app_process
    1084 0x000000002a003000 0x000000002a351000 rw-           0x0      0      0          0 [heap]
    1084 0x0000000040000000 0x000000004000e000 r-x           0x0     31      0        731 /system/bin/linker
    1084 0x000000004000e000 0x000000004000f000 r--           0x0      0      0          0
    1084 0x000000004000f000 0x0000000040010000 r--        0xe000     31      0        731 /system/bin/linker
    1084 0x0000000040010000 0x0000000040011000 rw-        0xf000     31      0        731 /system/bin/linker
    1084 0x0000000040011000 0x000000004001b000 rw-           0x0      0      0          0
    1084 0x000000004001b000 0x000000004001e000 r-x           0x0     31      0        630 /system/lib/liblog.so
    1084 0x000000004001e000 0x000000004001f000 r--        0x2000     31      0        630 /system/lib/liblog.so
    1084 0x000000004001f000 0x0000000040020000 rw-        0x3000     31      0        630 /system/lib/liblog.so
    1084 0x0000000040020000 0x0000000040065000 r-x           0x0     31      0        567 /system/lib/libc.so
    1084 0x0000000040065000 0x0000000040066000 ---           0x0      0      0          0
    1084 0x0000000040066000 0x0000000040068000 r--       0x45000     31      0        567 /system/lib/libc.so
    1084 0x0000000040068000 0x000000004006a000 rw-       0x47000     31      0        567 /system/lib/libc.so
    1084 0x000000004006a000 0x0000000040075000 rw-           0x0      0      0          0
    1084 0x0000000040075000 0x0000000040076000 r-x           0x0     31      0        474 /system/lib/libstdc++.so
    1084 0x0000000040076000 0x0000000040077000 r--           0x0     31      0        474 /system/lib/libstdc++.so
    1084 0x0000000040077000 0x0000000040078000 rw-        0x1000     31      0        474 /system/lib/libstdc++.so
    1084 0x0000000040078000 0x000000004008d000 r-x           0x0     31      0        537 /system/lib/libm.so
    1084 0x000000004008d000 0x000000004008e000 r--       0x14000     31      0        537 /system/lib/libm.so
```

9. Appendix 2

9.1. *Guide to dumping memory from Android emulator*

This guide is based on the guide provided by Volatility [10] with more detail and minor modifications.

Get the SDK

```
wget https://dl.google.com/android/adt/adt-bundle-linux-x86_64-20140702.zip
```

Unzip SDK

```
unzip adt-bundle-linux-x86_64-20140702.zip
```

Move to better path

```
mv adt-bundle-linux-x86_64-20140702.zip  /android-sdk
```

To be able to run 32-bit on 64-bit

```
sudo apt-get install libc6-i386 lib32stdc++6
```

Move to better path

```
mv android-ndk-r10c  /android-ndk
```

[10] https://github.com/volatilityfoundation/volatility/wiki/Android

Install required packages
`sudo apt-get install openjdk-7-jdk bison g++-multilib git gperf libxml2-utils curl`
Set up cache
`export USE_CCACHE=1` Set /bin in your $PATH `mkdir /bin`
`PATH= /bin:$PATH`
Download the repo tool and make it executable
`curl https://storage.googleapis.com/git-repo-downloads/repo > /bin/repo`
`chmod a+x /bin/repo`
Create repo dir
`mkdir /android-repo && cd /android-repo`
Set git git config if you have not done so already
`git config --global user.email "you@example.com"`
`git config --global user.name "Your Name"`
Initialize repo
`repo init -u https://android.googlesource.com/`
`platform/manifest`
Sync repo (this will take a long time and take up approximately 31 GB)
`repo sync`
set up environment
`. build/envsetup.sh`
Set some variables
`lunch full-eng`
Update the SDK
`android update sdk -u`
Before creating the AVD, you need the Image for the target we are using (4.2 API Level 17).
`android sdk`
Select ”ARM EABI v7a System Image” under API 17 and Install
Create avd
`android avd`
Set AVD name to ”myavd”, Device to Nexus 7 (2012) and Target to 4.2 (API Level 17). Select ”Display skin with hardware controls”. Make sure to set the SD-card size to larger than the amount of RAM, beacuse we are saving RAM to SD-card. We set 2 GB. Select Use host GPU.

Download the Kernel source
`git clone https://android.googlesource.com/kernel/goldfish.git /android-source`
`cd /android-source/`
Change branch to android-goldfish-2.6.29
`git checkout android-goldfish-2.6.29`
Before Compiling the kernel we need to set som variables
`export ARCH=arm`
`eexport SUBARCH=arm`
`export CROSS_COMPILE=arm-eabi-`
Create the config file
`make goldfish_armv7_defconfig`
Open the config file, and make sure the following is set:
`CONFIG_MODULES=y`
`CONFIG_MODULES_UNLOAD=y`
`CONFIG_MODULES_FORCE_UNLOAD=y`

Build the kernel. (when asked questions, just press enter for the default)
`make`
You can now start the emulator

```
emulator -avd myavd -kernel  /android-source/arch/arm/boot/zImage -show-kernel -verbose
```

Download LiME

```
git clone https://github.com/504ensicsLabs/LiME.git  /LiME
cd  /LiME/src
```

Edit the Makefile to correspond to this diff:

```
index 6de19f2..fc54b1f 100644
--- a/src/Makefile
+++ b/src/Makefile
@@ -24,15 +24,15 @@ obj-m := lime.o
 lime-objs := tcp.o disk.o main.o

 KVER ?= $(shell uname -r)
-
+KDIR_GOLDFISH := ~/android-source
+CCPATH := ~/android-ndk/toolchains/arm-linux-androideabi-4.8/prebuilt/linux-x86_64/bin
 PWD := $(shell pwd)

 .PHONY: modules modules_install clean distclen help

 default:
-       $(MAKE) -C /lib/modules/$(KVER)/build M=$(PWD) modules
-       strip --strip-unneeded lime.ko
-       mv lime.ko lime-$(KVER).ko
+       $(MAKE) ARCH=arm CROSS_COMPILE=$(CCPATH)/arm-linux-androideabi- -C $(KDIR_GOLDFISH) EXTRA_CFLAGS=-fno-pic M=$(PWD) modules
+       mv lime.ko lime-goldfish.ko

 debug:
        KCFLAGS="-DLIME_DEBUG" $(MAKE) -C /lib/modules/$(KVER)/build M=$(PWD) modules
```

Make the kernel module

```
make
```

Push the kernel module to the emulator

```
/android-sdk/sdk/platform-tools/adb push  /LiME/lime-goldfish.ko /sdcard/lime.ko
```

Start up a shell on the emulator

```
/android-sdk/sdk/platform-tools/adb shell
```

In the shell type:

```
insmod /sdcard/lime.ko "path=/sdcard/lime.dump format=lime"
```

You now have your memory dump in /sdcard/lime.dump

Transfer it yo your PC and you you can analyse it with volatility or other tools.