

A guide to Android memory forensics

Contents

1.	Introduction	3
2.	Preparing your host and the Android device	4
	2.1. Android Studio and adb	4
	2.2. Setting up an AVD	5
	2.3. Rooting	5
3.	Memory acquisition	6
	3.1. Compiling a Linux kernel	6
	3.1.1. Compiling and running the Goldfish kernel (AVD)	7
	3.1.2. Compiling the kernel of a physical Android device	10
	3.2. Compiling LiME	13
	3.3. Obtaining a memory image	13
	3.4. Replacing the stock boot image	15
4.	Android memory analysis techniques	18
	4.1. Volatility	18
	4.1.1. Creating a Volatility profile	19
	4.1.2. Memory analysis with Volatility	21
	4.2. Other analysis techniques	25
	4.2.1. Pattern matching	25
	4.2.2. File carving	29
5	Conclusion	30

1. Introduction

Android has been the best-selling operating system worldwide on smartphones since 2011 and as of May 2017 there are more than two billion monthly active Android devices.¹ As a consequence, the ability to analyse Android devices for forensic purposes has become increasingly important.

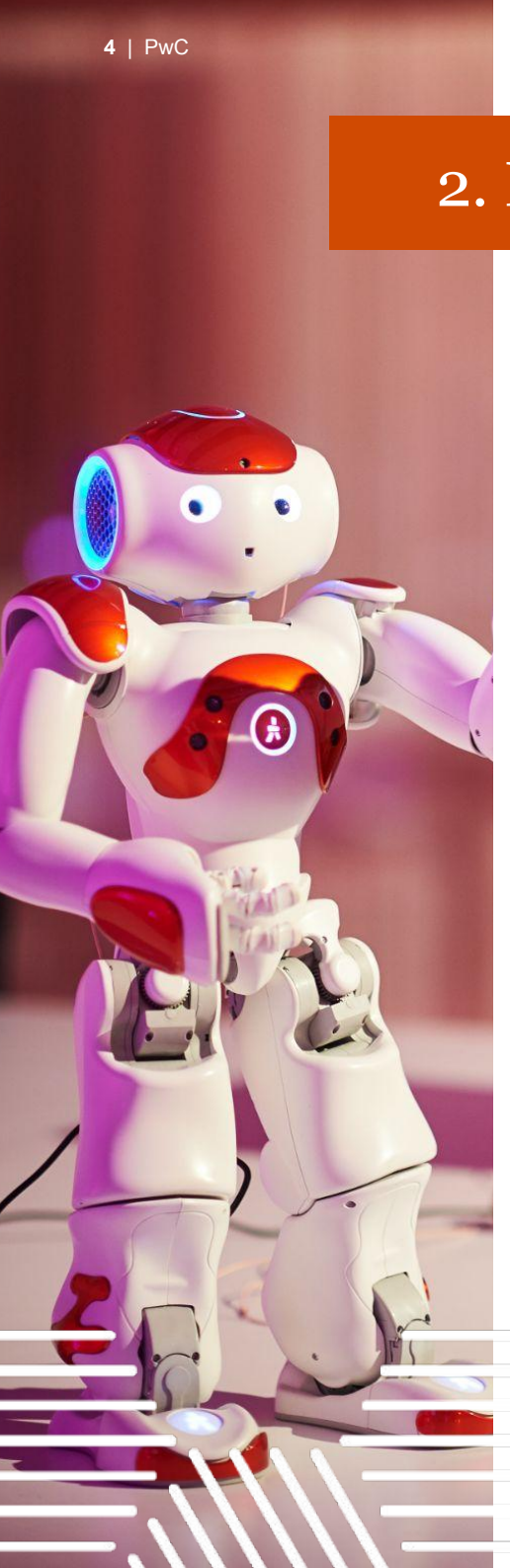
An important area in forensic analysis is the analysis of volatile memory. Volatile memory contains a wealth of information about the operating systems and userland software. Lots of this data, such as network artefacts and passwords, are not stored on non-volatile flash memory. Other artefacts, such as text messages, are often encrypted before they are stored on flash memory but still reside in plain text in memory. The analysis of volatile memory of Windows and Linux operating systems has become a common practice by forensic investigators and malware analysts, but this is not the case for Android devices. The analysis of volatile memory of Android still seems to be a blind spot for many analysts.

This article discusses a hands-on approach for software-based acquisition and analysis of volatile memory of Android devices. The term software-based relates to the method of acquisition, which is performed by executing software on a live Android device. This article also discusses the analysis of the memory image and demonstrates a few techniques to extract relevant system and user application data from memory. Following the same approach should enable you to identify and extract much more relevant information.

Note that the exact method to acquire and analyse Android memory depends on many factors including the device's manufacturer, Android version, kernel version and configuration, CPU architecture and more. This article therefore does not attempt to serve as a complete manual for all possible Android devices but applying the same approach should enable you to analyse memory of most of them (though in some cases with certain limitations).

¹ <https://twitter.com/Google/status/864890655906070529?s=20>

2. Preparing your host and the Android device



This section explains how to prepare your host computer and the Android device to be analysed (further referred to as the “target device”) for memory acquisition and analysis. The process that is explained in this article requires a Linux or macOS system as host due to the need to build Android source files, which is currently not supported on Windows. All examples in this article are applicable for a 64-bit Ubuntu 18.04 desktop system.

2.1. Android Studio and adb

Although not strictly necessary, it is recommended to install Android Studio on your host. Android Studio offers the ability to run emulated Android devices, which can greatly aid the testing of memory acquisition and analysis methods. Android Studio also comes with the Android Debug Bridge (adb).

This is a command-line tool that lets you communicate with an Android device that is connected to your host. The tool offers a Unix shell that you can use to run commands on the Android device, which will be used to acquire the contents from memory, as explained later in this article. Note that adb can also be installed via Aptitude, without the need to install Android Studio.

The instructions to install Android Studio are straightforward and can be found on <https://developer.android.com/studio/install>. On Linux, the installation can be started by extracting the installation ZIP file, which can be downloaded from the website of Android, to any location on your host. Then, Android Studio can be launched by executing `android-studio/bin/studio.sh`.

When executed for the first time, this will launch a setup wizard. In this wizard, the standard setup can be selected, which will install all components that are required to perform the memory acquisition and analysis actions that are explained in this article.

Note that on 64-bit Ubuntu, additional 32-bit libraries need to be installed:

```
$ sudo apt install libc6:i386  
libncurses5:i386 libstdc++6:i386  
lib32z1 libbz2-1.0:i386
```

To utilize adb tools with an Android Virtual Device (AVD), the user account of your host needs to be a member of the `plugdev` group and the default set of udev rules for Android devices should be installed:

```
$ usermod -aG plugdev $LOGNAME  
$ sudo apt install  
android-sdk-platform-tools-common
```

For more information about this, refer to <https://source.android.com/setup/build/initializing>.

To utilize adb on a physical Android device, USB debugging needs to be enabled. This is done in the developer options in the settings of the Android device. Note that on Android 4.2 and higher, the developer options are hidden by default.

The options can be revealed by tapping the “Build Number” in the settings 7 times. Note that the first time that a command is executed via adb, the Android device will display a prompt to ask whether your host is allowed to perform USB debugging.

2.2. Setting up an AVD

Before getting started with Android memory forensics, it can be useful to set up an Android Virtual Device (AVD) on your host for testing purposes. An AVD is a configuration that defines the characteristics of an Android device. This device can be simulated in the Android Emulator, which is part of Android Studio. The most important advantage of testing on an AVD is the flexibility to change components of the device such as Android version, kernel version, CPU architecture etc.

After the installation of Android Studio, a new AVD can be created in Android Studio’s AVD Manager. Here, an image without Google Play should be selected because these allow root access, which is not the case for images with Google APIs. When selecting an ARM image,

it is recommended to choose a 32-bit image because analysis of 64-bit ARM is not yet supported by the analysis tool that is discussed later (Volatility). For the examples in this article, an armeabi-v7a image was used. After selecting the image, the SD card should be configured in the advanced AVD settings to have a larger capacity than the RAM, to ensure that a full memory image can be stored on the SD card.

At this stage, the AVD can already be launched in the emulator. However, to make the AVD ready for testing, it should be launched with a kernel image that is compiled from its source code on your host. This way, an exact copy of the kernel that is running on the Android device will be available on your host. This copy will later in this article be used to cross compile two modules: one to acquire the contents of memory and the other to improve memory analysis capabilities. Eliminating any differences between the kernel on your host and the kernel on the Android device will enable you to get the best possible results.

2.3. Rooting

The memory acquisition that is explained in this article makes use of a loadable kernel module (LKM) to obtain the contents of memory. The module needs to be loaded into the kernel of the Android device, for which root privileges are required. Rooting the Android device is not within the scope of this article and the procedure differs for each device. However, there is plenty of documentation online on how to root various Android devices. The memory acquisition and analysis process that is documented as example in this article was performed on a Nexus 5 device, which was rooted using Nexus 5 CF-Auto-Root (<https://forum.xda-developers.com/google-nexus-5/orig-development/nexus-5-cf-auto-root-t2507211>).



3. Memory acquisition

In the context of forensic analysis, as much data as possible should be safeguarded.

Therefore, the goal is to acquire a physical memory image of Android devices. Note that various tools exist that can dump the address space of individual processes, such as Fridump (<https://github.com/Nightbringer21/fridump>), but these are not discussed in this article. Instead, this article will discuss the memory acquisition process with the Linux Memory Extractor (LiME). This tool can acquire physical memory from Linux devices and Linux-based devices such as Android (<https://github.com/504ensicsLabs/LiME>). It was first presented at Shmoocon in 2012 by Joe Sylve. LiME is a loadable kernel module (LKM) that performs the entire memory acquisition within the kernel, without context switches between userland and the kernel. This makes LiME forensically sound and minimizes discrepancies between the original contents in memory and the data in the memory image.

Because LiME is a kernel module, it needs to be compiled for the kernel version that is running on the target device. For devices that run a Linux operating system, this can be accomplished by compiling the code directly on the target device (although the best practice is to compile on a duplicate dummy system to avoid tampering with the investigated system). However, this approach is not suitable for Android devices because it is not feasible to compile code on the Android device directly. Instead, the LiME module should be compiled on a Linux or macOS host with a cross compiler. Before this can be done, the kernel of the target device first needs to be compiled on your host. This process is explained in the next section.

3.1. Compiling a Linux kernel

The Android operating system's kernel is based on the Linux kernel, so the compilation process is similar to that of a common Linux desktop or server kernel. To get started, an Android build environment needs to be set up on your host.

This entails the installation of software that is required to compile Android source code and kernels. A Linux or Mac system is required to do this, Windows is not currently supported. The process that is demonstrated in this article is applicable for a 64-bit Ubuntu 18.04 desktop host.

The most up-to-date instructions for Ubuntu and other operating systems are available on the website of Android (<https://source.android.com/setup/build/initializing>).

The following command installs the required packages on a 64-bit Ubuntu 18.04 desktop system:

```
$ sudo apt install git-core  
gnupg flex bison build-essential  
curl g++-multilib  
lib32ncurses5-dev lib32z1-dev  
libgl1-mesa-dev libxml2-utils  
xsltproc
```

Compiling the kernel of a physical Android device and an AVD is similar. However, the use case for the physical device and AVD in this article is different, which changes the approach. The goal for the physical Android device is to preserve its state as much as possible. Therefore, an identical kernel should be created on your host without tampering with the target device. This involves identifying and obtaining the kernel source code, kernel configuration and cross compiler that was used by the manufacturer of the device in order to recreate an identical (or similar) kernel on your host. For the AVD, the opposite approach is taken. Instead of recreating the original kernel of the AVD, any compatible kernel can be created and used to replace the original AVD kernel. Therefore, the approach for a physical device and an AVD are discussed separately in the next two sections.

3.1.1. Compiling and running the Goldfish kernel (AVD)

The Goldfish kernel was created by Google to be used with Android Studio's emulator. It contains additional functionality that enables the host of the emulator to interact with the AVD. Compiling the Goldfish emulator is a prerequisite for compiling the LiME module. The resulting kernel image can also be used to emulate the AVD that was created in section 2.2, which ensures that the LiME module is optimally compatible with the AVD. Additionally, compiling the kernel will yield the `System.map` file. This is one of the two files that are required to create a Volatility profile, which is needed to analyse the memory image with Volatility, as discussed later in this article.

The goldfish kernel can be cloned from [googlesource.com](https://android.googlesource.com/):

```
$ git clone
https://android.googlesource.com
/kernel/goldfish
```

After cloning the kernel, one of its branches needs to be checked out. Note that the current Android emulator

requires a Linux kernel of version 3.10 or higher. During the creation of this article, the goldfish 3.18 kernel was tested and confirmed to be compatible with the LiME module.

```
$ cd goldfish
$ git branch -a
$ git checkout
remotes/origin/android-goldfish-
3.18
```

After cloning the source code and performing a checkout, a compiler needs to be chosen to cross compile the source code with. The Android Open Source Project (AOSP) includes several compilers located in the directory `android-source/prebuilts/gcc/linux-x86-arm`. Alternatively, compilers can be downloaded from <https://android.googlesource.com>. If an ARM image was selected for the AVD, then an `arm-eabi-gcc` compiler should be used to compile the goldfish kernel. The process described in this article utilized the `arm-eabi-4.8` compiler:

```
$ git clone --depth=1
https://android.googlesource.com
/platform/prebuilts/gcc/linux-x86
/arm/arm-eabi-4.8
```

Before compiling the source code, some variables of the Makefile need to be configured. By default, the ARCH variable is set to the architecture of the device on which the make command is executed, which would likely be x86_64 when running a 64-bit Linux system. In this example, the kernel is compiled for an ARM AVD, so this variable needs to be overwritten. This can be done by either passing ARCH=arm to the make command or by setting ARCH as an environment variable:

```
$ export ARCH=arm
```

The variable CROSS_COMPILE also needs to be set. This variable specifies the common prefix of all executables used during compilation. These executables are located in the bin directory of the ARM compiler.

```
$ export  
CROSS_COMPILE=~/.Android/arm-eabi-4.8/bin/arm-eabi-
```

Inspection of the Makefile shows that it defines the full path to the required executable files. For instance, the full path to arm-eabi-gcc is defined by the line `CC = $(CROSS_COMPILE)gcc`.

To remove any previously generated files, navigate back to the kernel repository and execute `make clean`:

```
$ cd ~/goldfish  
$ make clean
```

The last step before compiling is to set up a working config. This is a configuration file that defines the features of the compiled kernel. For instance, a relevant option that is defined by this config file is whether the kernel should support the loading of kernel modules. Loadable module support is required in order to acquire a memory image with LiME, because LiME is a loadable kernel module that needs to be loaded into the kernel.

The kernel repository likely contains one or more config files, in this case located in the directory `goldfish/arch/arm/configs`. To obtain the exact config file that was used by the manufacturer to compile the kernel of the target device (the AVD in this case), the original config file can be extracted from the device:

```
$ ~/Android/Sdk/platform-tools/adb  
pull /proc/config.gz  
$ gunzip config.gz
```

Note that the original config file is not present on all Android devices. The original config file that was used for the compilation of the kernel by the manufacturer determined whether a copy of itself is made available on the target device via the option “Enable access to `.config` through `/proc/config.gz`”.

To utilize the config file, copy it to a file called `.config` in the kernel repository and run the `make menuconfig` command:

```
$ cp config ~/goldfish/.config  
$ make menuconfig
```

This will display a menu with the various kernel features that can be configured. The options “Enable loadable module support” and “Module unloading” should be enabled here, if this is not the case already. Finally, to compile the kernel, execute the following command in the kernel repository:

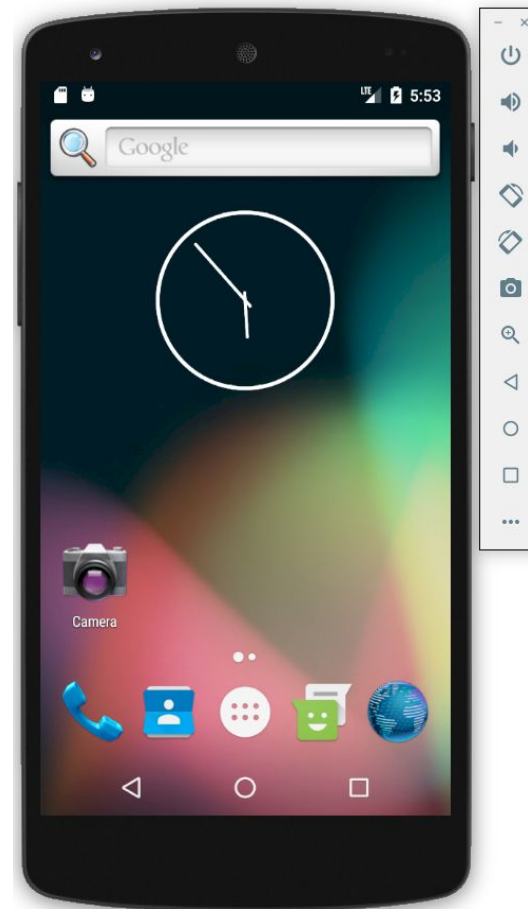
```
$ make modules_prepare  
$ make
```



The compilation will take several minutes. When successful, the kernel image (zImage) will be located in the directory arch/arm/boot. The compilation also generates the System.map file. This file is required to create a Volatility profile, which is explained later in this article.

The AVD can be launched with the new kernel image by launching the Android emulator from the command line with the option -kernel, followed by the path to the kernel image. Note that the directory ~/Android/Sdk gets created when launching Android Studio for the first time and running the standard setup.

```
$ cd ~/Android/Sdk/emulator
$ ./emulator -list-avds
$ ./emulator -avd Nexus_5_API_23
-kernel
~/goldfish/arch/arm/boot/zImage
-show-kernel -verbose
```



3.1.2. Compiling the kernel of a physical Android device

The kernel source code of physical Android devices can usually be downloaded from the developer or manufacturer's website. Simply searching the manufacturer's name of the device followed by "kernel source" online should lead you to a repository. While the steps detailed in this article will differ for each device, the approach remains the same.

The memory acquisition that is detailed in this article is applicable for a Nexus 5 device. This is an Android smartphone that was developed by Google and LG Electronics. The kernel source code of Google devices is hosted on <https://android.googlesource.com/kernel>. Some manufacturer websites enable you to find the kernel repository by looking for the device model number, but this is currently not the case here. However, the website contains a repository called "msm" which is described as "Kernel tree for Qualcomm chipsets". Online documentation states that the Nexus 5 contains a Qualcomm SoC (Qualcomm Snapdragon 800), so based on this information, the "msm" repository was selected.

```
$ git clone https://android.googlesource.com/kernel/msm
$ cd msm
```

The git command downloaded the complete msm kernel repository, including all branches. The LiME module should be compiled against the exact same kernel version that is running on the Android device. To know which version this is, the "Kernel version" section in the Android device's settings can be inspected. This section states the version and date of the kernel:

Kernel version

```
3.4.0-gcf10b7e
android-build@wpiv11.hot.corp.google.com #1
Mon Sep 19 22:14:08 UTC 2016
```

One possible approach to find the corresponding branch in the kernel repository, is to compare the version numbers and dates. An overview of branches and their corresponding dates can be listed by executing the following command from within the kernel repository:

```
$ git branch -avv | while read; do echo -e $(git log -1
--format=%ci $(echo "$REPLY" | awk '{print $2}' | perl -pe
's/\e\[?.*?[\@-~]//g') 2> /dev/null || git log -1
--format=%ci)" $REPLY"; done | sort -r | cut -d ' ' -f -1,4-
```

The following screenshot shows a snippet of the command output:

```
2016-10-06 remotes/origin/android-msm-angler-3.10-n-mr1-preview-1
2016-10-04 remotes/origin/android-msm-bullhead-3.10-n-mr1-preview-1
2016-09-20 remotes/origin/android-msm-seed-3.10-nougat
2016-09-19 remotes/origin/android-msm-shamu-3.10-marshmallow-mr1
2016-09-19 remotes/origin/android-msm-angler-3.10-nougat-mr0.5
2016-09-19 remotes/origin/android-msm-angler-3.10-nougat
2016-09-19 remotes/origin/android-msm-bullhead-3.10-nougat
2016-09-16 remotes/origin/android-msm-hammerhead-3.4-marshmallow-mr3
2016-09-13 remotes/origin/android-msm-sprat-3.10-marshmallow-mr1-wear-release
2016-09-12 remotes/origin/android-msm-anthias-3.10-marshmallow-mr1-wear-release
```

In this example, the operating system of the Nexus 5 device is Android 6.0.1 (Marshmallow), its kernel version is 3.4.0 and the kernel compilation date is 19 September 2016, so there is a clear match with the branch called `android-msm-hammerhead-3.4-marshmallow-mr3`.

```
$ git checkout -t
remotes/origin/android-msm-hammerhead-3.4-marshmallow-mr3
```

The next step is to find the right compiler to compile the kernel source code with. The compiler to be used depends on the CPU architecture of the target device and ideally you should use the same compiler that was used by the manufacturer. The CPU architecture can be identified by executing the `getprop` command with the option `ro.product.cpu.abi` on the target device via `adb`:

```
user@linux:~$ adb shell getprop ro.product.cpu.abi
armeabi-v7a
```

This command showed that the CPU architecture of the Nexus 5 is `armeabi-v7a`. Now, the version of the compiler that was used can be obtained by reading from `/proc/version`:

```
user@linux:~$ adb shell "cat /proc/version"
Linux version 3.4.0-gcf10b7e (android-build@wpiv11.hot.corp.google.com)
(gcc version 4.8 (GCC) ) #1 SMP PREEMPT Mon Sep 19 22:14:08 UTC 2016
```

From the information in the above screenshots it can be concluded that the manufacturer used the GNU Compiler Collection (GCC) version 4.8 to compile the kernel for an `armeabi-v7a` architecture. With this information, the corresponding compiler can be found on <https://android.googlesource.com>.

```
$ git clone --depth=1
https://android.googlesource.com/platform/prebuilts/gcc/linux-x86
/arm/arm-eabi-4.8
```



The next few steps are identical to the steps that were taken while setting up an AVD in the previous section, namely setting some environment variables and setting up a working config. For more details about these steps, please refer to section 3.1.1.

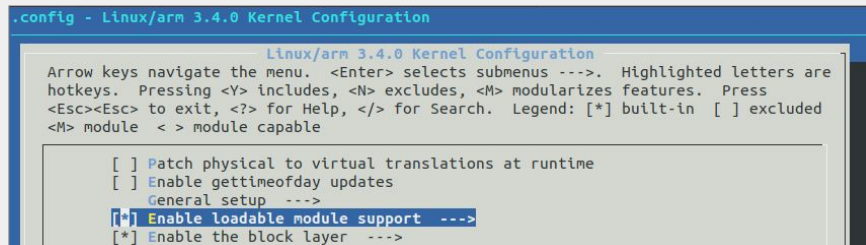
```
$ export ARCH=arm
$ export CROSS_COMPILE=~/.Android/arm-eabi-4.8/bin/arm-eabi-
```

In this case, the config file was not present on the Android device, likely because the kernel option “Enable access to `.config` through `/proc/config.gz`” was not enabled by the manufacturer. If the config file cannot be obtained from the device, a config file from the kernel repository can be used instead. The `msm` kernel repository contained 144 configuration files, located in the `arch/arm/configs` directory. The codename of the Nexus 5 (“Hammerhead”) indicated that the corresponding config file is called `hammerhead_defconfig`.

To inspect or change the options in the configuration file, copy it to the root directory of the repository and execute the `make_defconfig.sh` script with the configuration file as parameter:

```
$ cd ~/Android/msm
$ cp arch/arm/configs/hammerhead_defconfig ~/Android/msm/
$ ./make_defconfig.sh hammerhead_defconfig
```

This will present a menu with the kernel configuration options. Here, the options “Enable loadable module support” and “Module unloading” are of interest. For some default kernels that are shipped with Android devices, these options were not enabled by the manufacturer. This hinders the memory acquisition process, because the LiME module cannot be loaded into the stock kernel. A possible workaround is discussed later in this article. For this workaround, a kernel image which does enable module loading is required. Therefore, it is recommended to enable both options now, in order to already obtain a suitable kernel image for this workaround. Enabling these options is not needed if you know that the kernel of the target device already supports module loading.



```
.config - Linux/arm 3.4.0 Kernel Configuration
Linux/arm 3.4.0 Kernel Configuration
Arrow keys navigate the menu. <Enter> selects submenus ---. Highlighted letters are
hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press
<Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [ ] excluded
<M> module < > module capable

[ ] Patch physical to virtual translations at runtime
[ ] Enable gettimeofday updates
  General setup --->
[*] Enable loadable module support --->
[*] Enable the block layer --->
```

Upon exiting and saving the configuration menu, the changes will be written to `.config`.

Finally, to compile the kernel, execute the following commands from within the kernel repository:

```
$ make modules_prepare
$ make
```

The compilation will take several minutes. When successful, the kernel image will be located at `arch/arm/boot/zImage`. The directory `arm` in this example represents the architecture of the kernel image and will differ based on the architecture. The compilation also generates the `System.map` file. This file is required to create a Volatility profile, which is explained later in this article. Note that in this case a `zImage-dtb` image was created, which is a kernel image with a device tree blob (DTB) appended to it.



3.2. Compiling LiME

The first step to compile LiME is to clone its source code onto your host.

```
$ git clone --depth=1
https://github.com/504ensicsLabs
/LiME.git
$ cd LiME/src
```

Now, the Makefile needs to be edited to include the path to the compiled target kernel (KDIR), the path to the cross compiler which was used to compile the kernel source code (CCPATH), the target kernel version (KVER) and its architecture (ARCH).

```
$ nano Makefile
```

```
obj-m := lime.o
lime-objs := tcp.o disk.o main.o hash.o deflate.o

KVER ?= 3.4.0
KDIR ?= /home/user/Android/msm/

PWD := $(shell pwd)
CCPATH := /home/user/Android/arm-eabi-4.8/bin/

.PHONY: modules modules_install clean distclean debug

default:
    $(MAKE) -C $(KDIR) M="$(PWD)" modules
    $(CCPATH)/arm-eabi-strip --strip-unneeded lime.ko
    mv lime.ko lime-$(KVER).ko
```

The LiME module can now be compiled by executing the following commands from the root directory of the LiME repository.

```
$ make clean
$ make
```

This generated a file called `lime-3.4.0.ko`, where 3.4.0 will match the value that was assigned to KVER in the Makefile.

3.3. Obtaining a memory image

At this stage, the only thing left to do to acquire the contents of memory is to load the LiME module in the kernel of the target device. The module can be copied to the target device via `adb`'s push command:

```
$ ~/Android/Sdk/platform-tools/adb
push ~/Android/LiME/src/
lime-goldfish.ko
/sdcard/lime-goldfish.ko
```

The AVD used as example in this article failed to copy the file because its file system was read-only:

```
adb: error: failed to copy
<source> to <dest>: Read-only
file system.
```

This can be solved by remounting the filesystem as read-write, as root. In this example, root access was granted automatically upon executing `adb shell`, but if this is not the case then root access can be acquired by running the `su` command (granted that the target device is an AVD without Google APIs or a physical device that was rooted).

```
$ ~/Android/Sdk/platform-tools/adb
shell
root@generic:/ # su
root@generic:/ # mount -o
rw,remount rootfs /
```

After copying the LiME module to the target device, open a shell with root privileges on the device (if not done already) and load the LiME module via the `insmod` command.

```
$ ~/Android/Sdk/platform-tools/adb
shell
root@generic:/ # su
root@generic:/ # cd /sdcard/
root@generic:/ # insmod
lime-goldfish.ko
"path=android.lime format=lime"
```

The output filename is passed via the `path` parameter. The format can either be `raw`, `padded` or `lime`. The `raw` format will store data of acquired sections after each other in the output file. However, if certain regions cannot be acquired, this means that the offset of consecutive regions in the memory image will no longer match their original offset in physical memory. This can be avoided by using the `padded` format, which adds zeros to the output file to replace any regions that cannot be

captured. The `lime` format also keeps track of the original offsets of sections, but by storing this information as metadata in the memory image. These offsets are then used by compatible analysis tools to reconstruct the original memory layout.

Upon loading the module into the target kernel with the `insmod` command, the module will immediately start writing the content of memory to a file. The file can then be copied to your host with `adb`'s pull command:

```
$ ~/Android/Sdk/platform-tools/adb
pull /sdcard/android.lime
```

Note that the memory acquisition with LiME can also be done over a network. This may be preferred when the unallocated space of the Android device's storage should not be overwritten for forensic purposes. Instead of acquiring the data via a physical network, it can be acquired via `adb`'s forward command, which sets up port forwarding. The following example sets up forwarding of the Android device's port 4444 to the host's port 4444:

```
user@linux:~$
~/Android/Sdk/platform-tools/adb
forward tcp:4444 tcp:4444
```

LiME can then be instructed to send data to port 4444 by passing the protocol and port number instead of a filename to the `path` parameter:

```
root@generic:/ # insmod lime.ko
"path=tcp:4444 format=lime"
```

The data can then be received and saved to a file on your host with `netcat`:

```
$ nc localhost 4444 > memory.lime
```

By following the procedure described above, it was possible to acquire a full memory image of the AVD that was emulated on the host. Inspection of the memory image with the `strings` utility revealed valid data such as app names, URLs and more.

At this stage, the memory acquisition process should be complete for any device of which the kernel supports module loading. Some devices do not support this, and this causes an additional obstacle for the memory acquisition process, as was the case for the Nexus 5 device. The stock kernel of this device did not enable module loading and attempting to insert the LiME module therefore returned the following error:

```
root@hammerhead:/sdcard # insmod lime-3.4.0.ko "path=nexus5.lime format=lime"
insmod: failed to load lime-3.4.0.ko: Function not implemented
```



This is possible to bypass by replacing the original kernel image of the target device with a custom kernel image that does support module loading. Such a kernel image was already created in section 3.1.

A major drawback to replacing the original kernel image is that the target device needs to reboot in the process. This means that the original content of memory will be lost. Therefore, this workaround is less suitable to perform a forensic analysis of a suspect's Android device. However, it may still be useful to inspect memory after a reboot, for instance to observe persistent malware on the device.

The next section describes the process to create and flash a new boot image. This is only needed if the memory image could not be acquired at this point because the stock kernel of the device does not enable module loading. The next section can therefore be skipped if the memory image was already acquired at this stage.

3.4. Replacing the stock boot image

To replace the stock kernel on the target device with a custom compiled kernel image, a new boot image needs to be created. A boot image consists of a kernel image and a ramdisk.

To keep the target device as close to the original state as possible, it is recommended to extract and alter the original boot image from the target device. The following commands copy the boot image of a Nexus 5 to its shared storage (sdcard) and copy it via adb to the host.

Host

```
$ ~/Android/Sdk/platform-tools/adb
shell
```

Nexus 5

```
shell@hammerhead:/ $ su
root@hammerhead:/ # cp
/dev/block/platform/msm_sdcc.1/b
y-name/boot /sdcard/boot.img
```

Host

```
$ ~/Android/Sdk/platform-tools/adb
pull /sdcard/boot.img boot.img
```

The recommended way to modify the boot image is by using the CyanogenMod tools called `mkbooting` and `unpackbooting`, which can be cloned from the `android_system_core` repository on Github:

```
$ git clone --depth=1
https://github.com/CyanogenMod/a
ndroid_system_core.git
```

The `mkbooting` and `unpackbooting` tools can optionally be copied to `/usr/bin` to add them to a location that is in `$PATH`:

```
$ sudo cp
android_system_core/mkbooting/mk
booting /usr/bin
$ sudo cp
android_system_core/mkbooting/un
packbooting /usr/bin
```



To unpack the original boot image, run the `unpackbootimg` tool with the filename of the boot image passed with the `-i` parameter:

```
user@linux:~/Android/unpacked-boot.img$ unpackbootimg -i boot.img
Android magic found at: 0
BOARD_KERNEL_CMDLINE console=ttyHSL0,115200,n8 androidboot.hardware=hammerhead user_debug=31 maxcpus=2 msm_watchdog_v2.enable=1
BOARD_KERNEL_BASE 00008000
BOARD_RAMDISK_OFFSET 02900000
BOARD_SECOND_OFFSET 00f00000
BOARD_TAGS_OFFSET 02700000
BOARD_PAGE_SIZE 2048
BOARD_SECOND_SIZE 0
BOARD_DT_SIZE 0
```

This command extracts the components of the boot image and writes values such as offsets to new files on disk. Note that the values that were written to `boot.img-ramdisk_offset`, `boot.img-second_offset` and `boot.img-tags_offset` need to be edited to contain the prefix `0x`. Otherwise, the values won't be interpreted as hexadecimal values by the `mkbootimg` tool.

Now, all components can be packed into a new boot image in which the original kernel image is replaced by the kernel image that was compiled from source in section 3.1.

```
user@linux:~/Android/unpacked-boot.img$ mkbootimg --kernel zImage-dtb --ramdisk ./boot.img-ramdisk.gz
--second ./boot.img-second --cmdline "$(cat ./boot.img-cmdline)" --base "$(cat ./boot.img-base)" --p
agesize "$(cat ./boot.img-pagesize)" --dt ./boot.img-dt --ramdisk_offset "$(cat ./boot.img-ramdisk_of
fset)" --second_offset "$(cat ./boot.img-second_offset)" --tags_offset "$(cat ./boot.img-tags_offset)
" --output new-boot.img
```

The `--kernel` parameter points to the kernel image that was created in section 3.1 (`zImage-dtb`). Note that instead of copying all offset values into this command, the values are inserted in the command by reading the files created by `unpackbootimg`. The output filename is passed with the `--output` parameter (`new-boot.img`).

```
user@linux:~/Android/unpacked-boot.img$ file new-boot.img
new-boot.img: Android bootimg, kernel (0x8000), ramdisk (0x2900000), page size: 2048, cmdline (consol
e=ttyHSL0,115200,n8 androidboot.hardware=hammerhead user_debug=31 maxcpus=2 msm_watchdog_)
```


Booting the Android device from the new boot image, or permanently overwriting the original boot image with the new image, can be done with the fastboot tool. This tool can be installed via Aptitude:

```
$ sudo apt install fastboot
```

Before utilizing fastboot, the target device's bootloader needs to be accessed. This is done by powering down the device and then pressing a certain combination of buttons. The combination differs based on the device, but online documentation specifies the combination for various devices. To enter the bootloader on the Nexus 5, the power button and volume down button had to be pressed and held at the same time.

After entering the bootloader, the Android device needs to be connected via USB cable to the host. Then, the device can be instructed to boot from the new boot image:

```
$ sudo fastboot boot
new-boot.img
```

Now, the Android device should boot just like before and there should not be any noticeable difference, as the only thing that was changed is the ability to load kernel modules. Note that the above fastboot command did not overwrite the original boot image of the Android device. Therefore, if anything on the Android device would malfunction, it is possible to restore the device to its original state by simply rebooting the device.

After booting the Nexus 5 from the new boot image, it was possible to acquire a memory image with LiME by performing the steps that were explained in the previous section.



4. Android memory analysis techniques

A physical memory image contains application and kernel data in an unstructured manner. This section explains the basics of how to make sense of the data in the memory image.

The memory contains a wealth of data related to the operating system and user applications. User application data such as text messages, browser history and pictures can be extracted using string analysis and file carving techniques, as demonstrated in section 4.2. To extract general system information such as running processes and network connections, the Volatility framework is recommended.

4.1. Volatility

Volatility is an open source framework that is capable of extracting and analysing data from Windows, Linux and macOS memory images (<https://github.com/volatilityfoundation/volatility>). The analysis of 32-bit ARM Linux-based devices, such as Android, is also supported. Note that at the time of writing, Volatility does not yet support the analysis of memory images of 64-bit ARM devices. One of the co-authors of Volatility, Andrew Case, stated in September 2018 that this functionality is not yet implemented due to low demand.

Volatility's functionality mostly consists of plugins. A Volatility plugin is essentially a Python script that extends Volatility and that is dedicated to extracting specific information from a memory image, such as information on running processes, network connections, handles and so on.

Volatility can be installed via APTitude, but it is recommended to run the Python source code from Github instead, to have the latest version. Before analysing Android memory with Volatility, a profile needs to be created. This is explained in the next section.

4.1.1. Creating a Volatility profile

A Volatility profile is a file that contains information about a specific kernel, which is needed by Volatility to find and reconstruct data from a memory image in a structured format. Volatility includes pre-made Windows profiles, which means that Windows memory images can be analysed without having to create a profile. However, Volatility does not include any Linux profiles. The reason for this is that there are too many different Linux kernel versions to include a profile for each version. Therefore, it is generally up to the analyst to create a Volatility profile for analysing Linux or Android memory images.

A Volatility profile for Linux is a ZIP archive that contains two files: `System.map` and `module.dwarf`.

`System.map` contains a lookup between all symbol names from the Linux kernel image and their addresses in memory. This `System.map` file is generated during the compilation of the kernel. The following screenshot shows the first few lines (out of more than 80,000) of the `System.map` of the stock Nexus 5 kernel:

```
00000020 A cpu_v7_suspend_size
c0004000 A swapper_pg_dir
c0008000 T _text
c0008000 T stext
c000804c t __create_page_tables
c0008100 t __turn_mmu_on_loc
c000810c t __vet_atags
c0100000 T __exception_text_start
c0100000 T _stext
c0100000 T do_undefinstr
```

The kernel source code of the target device was already compiled in section 3.1., which means that the `System.map` file should already be present in the root directory of the kernel repository.

The second file that is part of the Volatility profile is called `module.dwarf`. This file contains a collection of VTypes. These are structure definitions used by Volatility to represent C data structures (used by Linux kernel) in Python (used by Volatility). Below is a simplified example of how a C data structure is represented as a VType.

C		VTypes (Python)
<pre>struct proc { int pid; char name[10]; char * cmd_line; };</pre>	→	<pre>'proc' : [18, { 'pid' : [0, ['int']], 'name' : [4, ['array', 10, ['char']]], 'cmd_line' : [14, ['pointer', ['char']] }]</pre>

The name of the structure (`proc`) becomes the name of a dictionary key, which contains the members along with their types and offset from the base of the structure (in this example an integer, a character array and a pointer to a string).

The `module.dwarf` file is obtained by compiling a dummy kernel module that declares members of all types needed by Volatility for the target kernel, and then extracting the DWARF symbols from the dummy module with the `dwarfdump` tool. The source code of this module is included in the repository of Volatility and is located at `volatility/tools/linux/module.c`. Note that compiling the module will automatically instruct the `dwarfdump` tool to extract DWARF symbols from the resulting module, so `dwarfdump` should be installed first:

```
$ sudo apt install dwarfdump
```

The next step is to clone Volatility from Github and to edit the Makefile of the dummy module (`module.c`), which is located under `volatility/tools/linux`.

```
$ git clone --depth=1
https://github.com/volatilityfoundation/volatility.git
$ cd ~/volatility/tools/linux
```

The Makefile needs to be edited to include the path to the compiled target kernel (`KDIR`), the path to the cross compiler which was used to compile the kernel source code (`CCPATH`), the target kernel version (`KVER`) and its architecture (`ARCH`). Note that in the following screenshot the variable `CROSS_COMPILE` is defined as `CCPATH` followed by `arm-eabi-`. This may need to be changed to match the common prefix of all executables located in the `bin` directory of the cross compiler.

```
$ nano Makefile
```

```
obj-m += module.o
KDIR := /home/Android/msm
CCPATH := /home/Android/arm-eabi-4.8/bin
KVER ?= 3.4.0

-include version.mk

all: dwarf

dwarf: module.c
    $(MAKE) ARCH=arm CROSS_COMPILE=$(CCPATH)/arm-eabi- -C $(KDIR) CONFIG_DEBUG_INFO=y M=$(PWD) modules
    dwarfdump -di module.ko > module.dwarf
```

Now, the dummy module can be compiled by executing `make`. This will produce the module called `module.ko` and, as instructed by the Makefile, the `dwarfdump` tool will extract DWARF symbols from this module and write them to a new file called `module.dwarf`. The following screenshot shows the start of the `module.dwarf` file that was created for the Nexus 5.

```
.debug_info
<0><0x0+0xb><DW_TAG_compile_unit> DW_AT_producer<GNU C 4.8 -mlittle-endian -marm -mapcs -mno-sched-prolog -mabi=aap
cs-linux -mno-thumb-interwork -march=armv7-a -mfloat-abi=soft -mfpu=vfp -g -O2 -p -fno-strict-aliasing -fno-common
-fno-delete-null-pointer-checks -fno-dwarf2-cfi-asm -fstack-protector -funwind-tables -fno-omit-frame-pointer -fno-
```

The final step to create a Volatility profile for Linux is to add the `System.map` and `module.dwarf` files into a ZIP archive.

```
$ zip Nexus5.zip module.dwarf System.map
```

The creation of the Volatility profile is now complete and the usage of this profile with Volatility is demonstrated in the next section. Note that in some cases, the profile might not be compatible with the stock kernel that is running on the target device. If this is the case, Volatility will throw the error “No suitable address space mapping found” when attempting to run any plugin. This indicates that most likely there is a discrepancy between the kernel that is running on the target device and the kernel that was cross compiled on your host. This discrepancy could be caused by using different compilation options than the manufacturer. The kernel configuration options which were used during kernel compilation may drastically change the resulting kernel image. Possible solutions include cross compiling the kernel source code again with different options (different `.config`) or flashing the kernel that was compiled on your host to the target device in order to eliminate any discrepancies.



4.1.2. Memory analysis with Volatility

In general, Volatility requires three command line parameters: the filename of the memory image (`--filename` or `-f`), the name of the profile to be used (`--profile`), and the plugin to be executed (specified without any prefixing parameter). When analysing non-Windows memory images, the directory that contains the Volatility profile (the ZIP file created in the previous section) also needs to be passed to Volatility via the `--plugins` parameter. This directory may also contain third-party Volatility plugins (Python files), which expand the functionality of Volatility. Note that the `--plugins` parameter must be the first parameter that is passed to Volatility, otherwise it will not be processed.

Before the full Volatility command can be constructed, the name of the profile that was created in the previous section needs to be identified. This can be done by running Volatility with parameter `--info`.

```
$ python2 vol.py --plugins=~/.Android/profiles --info
```

```
Profiles
-----
LinuxNexus5ARM      - A Profile for Linux Nexus5 ARM
VistaSP0x64         - A Profile for Windows Vista SP0 x64
VistaSP0x86         - A Profile for Windows Vista SP0 x86
VistaSP1x64         - A Profile for Windows Vista SP1 x64
VistaSP1x86         - A Profile for Windows Vista SP1 x86
```



Upon running this command, Volatility will parse the content of the profiles directory and include a list of valid profiles in its output. Any custom profiles should be listed in the output above the built-in Windows profiles. The name of the custom profile is based on the filename of the ZIP archive. In this example, the ZIP archive was called Nexus5.zip, which caused the text “Nexus5” to be present in the profile name. Volatility detected that the profile is suitable for a memory image of a system with a Linux kernel and ARM architecture, and added this information to the profile name. This resulted in the profile name LinuxNexus5ARM. This is the profile name that must be passed to Volatility via the `--profile` parameter in this example.

Because Android is a Linux-based operating system, the Linux plugins must be used to analyse Android memory. The names of Linux-compatible plugins are prefixed by `linux_`. An overview of these plugins is included in the info output of the previous command.

The following screenshot shows an example Volatility command which executes the `linux_pslist` plugin. This plugin extracts a list of running processes from the memory image. Note that the screenshot is cropped to only include the first few processes.

```
user@linux:~$ python2 volatility/vol.py --plugins=profiles --profile=LinuxNexus5ARM -f Nexus5.lime linux_pslist
Volatility Foundation Volatility Framework 2.6.1
Offset      Name                Pid      PPid      Uid        Gid      DTB        Start Time
-----
0xee068000  init                 1         0         0          0        0x3556c000 2020-11-12 14:33:09 UTC+0000
0xee068500  kthreadd             2         0         0          0        ----- 2020-11-12 14:33:09 UTC+0000
0xee068a00  ksoftirqd/0         3         2         0          0        ----- 2020-11-12 14:33:09 UTC+0000
0xee068f00  kworker/0:0          4         2         0          0        ----- 2020-11-12 14:33:09 UTC+0000
0xee069400  kworker/0:0H        5         2         0          0        ----- 2020-11-12 14:33:09 UTC+0000
0xee069900  kworker/u:0          6         2         0          0        ----- 2020-11-12 14:33:09 UTC+0000
0xee069e00  kworker/u:0H        7         2         0          0        ----- 2020-11-12 14:33:09 UTC+0000
0xee06a300  migration/0         8         2         0          0        ----- 2020-11-12 14:33:09 UTC+0000
0xee06c100  khelper              13        2         0          0        ----- 2020-11-12 14:33:09 UTC+0000
0xee06c600  netns                14        2         0          0        ----- 2020-11-12 14:33:09 UTC+0000
0xee06cb00  kworker/0:1         15        2         0          0        ----- 2020-11-12 14:33:09 UTC+0000
```

```
GNU nano 4.8  volatilityrc
[DEFAULT]
PLUGINS=/home/user/Android/profiles
LOCATION=file:///home/user/Android/Nexus5.lime
PROFILE=LinuxNexus5ARM
```

Note that Volatility enables you to save command line parameters in a configuration file called `volatilityrc`. This file will be parsed when stored in the current working directory. The configuration file on the left was created to shorten the following commands in this article. Additionally, the alias `vol.py` was created for the command `python2 ~/volatility/vol.py`.

There are currently over 70 Linux plugins. This article demonstrates only a couple of those.

linux_arp

The `linux_arp` plugin extracts ARP cache from the memory image. ARP cache keeps track of MAC addresses that were resolved via the Address Resolution Protocol (ARP). This cache can give a view on systems that the mobile device recently connected to within the same subnet.

```
user@linux:~$ vol.py linux_arp
Volatility Foundation Volatility Framework 2.6.1
[ff02::1] at 33:33:00:00:00:01 on wlan0
[fe80::9e97:26ff:fe25:5a3c] at 9c:97:26:25:5a:3c on wlan0
[::] at 00:00:00:00:00:00 on lo
[192.168.1.1] at 9c:97:26:25:5a:3c on wlan0
[239.255.255.250] at 01:00:5e:7f:ff:fa on wlan0
```

In the above screenshot, the IP address 192.168.1.1 is seen. This IP address typically corresponds to the IP address of the default gateway. This information could help to indicate that the Android device connected to a particular wireless access point, by matching the MAC address in this command output to the MAC address of the corresponding access point. Other artefacts such as SSIDs of saved access points, maps application data and GPS data may further help to find the location of that wireless access point.

linux_route_cache

The `linux_route_cache` plugin extracts routing cache from the memory image. This cache can give a view on IP addresses that the Android device connected to. Connections to these IP addresses could have been initiated by any software on the device including the OS, system applications and user applications.

```
user@linux:~$ vol.py linux_route_cache | grep -v lo
Volatility Foundation Volatility Framework 2.6.1
-----
Interface      Destination      Gateway
wlan0          107.167.123.37   192.168.1.1
wlan0          107.167.123.6    192.168.1.1
wlan0          152.195.132.202  192.168.1.1
wlan0          159.180.84.2     192.168.1.1
wlan0          169.48.4.165    192.168.1.1
wlan0          172.217.168.202  192.168.1.1
wlan0          172.217.168.206  192.168.1.1
wlan0          172.217.168.234  192.168.1.1
wlan0          172.217.168.238  192.168.1.1
```

Note that connections by the loopback interface were filtered out of the above example. To receive more information about the network connections, for instance which process they belonged to, the `linux_netstat` plugin can be used. This plugin lists all open sockets, but unfortunately the plugin's output was empty for every Android memory sample tested during the creation of this article.

linux_enumerate_files

The `linux_enumerate_files` plugin can find files and directories in memory by identifying and parsing file system structures. The plugin outputs the inode address, inode number and file path.

```
user@linux:~$ vol.py linux_enumerate_files
Volatility Foundation Volatility Framework 2.6.1
Inode Address Inode Number      Path
-----
0xec5b3dc0    324 /mnt/runtime/write/emulated/0/DCIM/Camera/IMG_20190123_134613.jpg
0xeb440000    319 /mnt/runtime/write/emulated/0/DCIM/Camera/IMG_20190123_134132.jpg
0xe1605c00    318 /mnt/runtime/write/emulated/0/DCIM/Camera/IMG_20190123_134122.jpg
0xe1605dc0    317 /mnt/runtime/write/emulated/0/DCIM/Camera/IMG_20190123_134113.jpg
```

The Linux kernel uses a page cache to save content of files that are loaded in memory to improve performance. The `linux_find_file` plugin can extract files from this page cache. However, utilizing this plugin to extract files from the memory of several AVD devices and a Nexus 5 Android device gave poor results. More specifically, most of the files that were extracted could not be opened by the appropriate applications (based on file extension) and the `file` utility did recognize most of the extracted files as a particular file type, which indicates that the extracted files were corrupt. Luckily it is still possible to extract valid files from memory by using file carving techniques instead, which is demonstrated in the next section.

linux_lsof

The `linux_lsof` plugin displays a list of open files, similar to the `lsof` command on Linux. This plugin can give more context about the files in memory because it shows which process opened which file. For instance, if an image was opened by a messaging application, this could indicate that the image was sent or received via the application. Note that there are other reasons for a messaging application to open images, for instance to let the user browse the images on the device to select one or more to send. The directory in which the image is stored and the conversation data of the messaging application could be further inspected to obtain an indication of whether the image was sent or received.

```
user@linux:~$ vol.py linux_lsof -p 1519
Volatility Foundation Volatility Framework 2.6.1
Offset          Name                Pid    FD    Path
-----
0x00000000eae4100 m.android.phone    1519   5    /system/framework/framework-res.apk
0x00000000eae4100 m.android.phone    1519   6    /system/framework/core-libart.jar
0x00000000eae4100 m.android.phone    1519  14    /system/priv-app/TeleService/TeleService.apk
0x00000000eae4100 m.android.phone    1519  15    /system/priv-app/TelephonyProvider/TelephonyProvider.apk
0x00000000eae4100 m.android.phone    1519  16    /data/data/com.android.providers.telephony/databases/HbpcdLookup.db
0x00000000eae4100 m.android.phone    1519  17    /data/data/com.android.providers.telephony/databases/telephony.db
0x00000000eae4100 m.android.phone    1519  35    /data/data/com.android.providers.telephony/databases/mmssms.db
0x00000000eae4100 m.android.phone    1519  36    /system/app/Hangouts/Hangouts.apk
0x00000000eae4100 m.android.phone    1519  41    /data/app/com.google.android.apps.messaging-1/base.apk
0x00000000eae4100 m.android.phone    1519  42    /data/app/org.thoughtcrime.securesms-1/base.apk
0x00000000eae4100 m.android.phone    1519  47    /system/app/Stk/Stk.apk
```



4.2. Other analysis techniques

Volatility currently does not have any plugins to extract user data from Android applications. Each application stores data differently in memory and due to the vast number of Android applications, many of which are regularly updated, it would be impractical to write and maintain plugins that extract application data. However, relevant application data can still be obtained by using general string analysis and linear file carving techniques.

Even though it is possible to extract artefacts directly from a physical memory image, it is in most cases recommended to search and extract data from the process memory instead. The main reason for this is that the virtual address space of a process is continuous, unlike the data in a physical memory image. More specifically, data that crosses page boundaries in the virtual address space may be split across multiple locations in physical memory. As a result, signatures might fail to match these patterns in physical memory. This is especially relevant to keep into account when carving files from memory that are larger than the page size, which is typically 4 kB.

Another advantage of analysing process memory is that you know which process the matched patterns belong to. This context is important to determine the meaning and relevance of data in memory and would be missed when directly analysing the physical memory image. Another advantage of analysing process memory is that data from irrelevant processes is filtered out, which potentially results in less irrelevant hits.

Analysing physical memory can still be useful in certain cases, for instance when looking for data related to terminated processes. The virtual address space of terminated processes cannot be reconstructed, but remnant data in freed pages that have not yet been overwritten can still be recovered from physical memory.

4.2.1. Pattern matching

This section demonstrates several examples of application and system data that could be extracted from a Nexus 5 device via pattern matching, in order to demonstrate the general methodology and techniques that can be used. There are several possible approaches to matching patterns in memory. One approach is to use Volatility's `linux_yarascan` plugin to find signatures that match any textual or binary patterns in the process memory of one or more processes. Relevant data to look for may include text messages, browser history, call logs, passwords etc. The following example shows how the `linux_yarascan` plugin is used to find HTTP Host headers in the memory of the Opera browser application. This can reveal browser history even after the history was deleted from the flash memory.

```
user@linux:~$ vol.py linux_yarascan -p 10471 -Y 'Host: '
Volatility Foundation Volatility Framework 2.6.1
Task: m.opera.browser pid 10471 rule r1 addr 0x80f2b400
0x80f2b400 48 6f 73 74 3a 20 77 77 77 2e 65 61 72 74 68 72 Host:.www.earthr
0x80f2b410 61 6e 67 65 72 73 2e 63 6f 6d 0d 0a 43 6f 6e 6e angers.com..Conn
0x80f2b420 65 63 74 69 6f 6e 3a 20 6b 65 65 70 2d 61 6c 69 ection:.keep-all
0x80f2b430 76 65 0d 0a 4f 72 69 67 69 6e 3a 20 68 74 74 70 ve..Origin:.http
0x80f2b440 73 3a 2f 2f 77 77 77 2e 65 61 72 74 68 72 61 6e s://www.earthran
0x80f2b450 67 65 72 73 2e 63 6f 6d 0d 0a 00 00 00 00 00 00 gers.com.....
```

In this example, only the process memory of the browser was searched by passing its PID with the `-p` option. Omitting this option will cause the plugin to search all processes but will take considerably longer. However, this could be a good way to determine in which process(es) a certain pattern resided. By default, the `linux_yarascan` plugin only searches the userland memory, but the kernel memory can be included by passing the `-K` option. In this example, the YARA rule was passed as a string with the `-Y` option, but it is also possible to pass a YARA rule file with the `-y` option instead.

The same results can be obtained by extracting the process memory to disk with the `linux_dump_map` plugin and searching the extracted memory regions with utilities such as `yara`, `strings` and `grep`. The `linux_dump_map` plugin works by identifying the starting and ending addresses of process' memory ranges and dumping each page within those ranges to disk. The addresses obtained by Volatility match the addresses that would be obtained on a live system by reading from `/proc/<pid>/maps`. Passing the PID with the `-p` option to the `linux_dump_map` plugin will cause all mappings of the specified process to be dumped. The `linux_dump_map` plugin also requires an output directory to be passed with the `-D` parameter.

```
$ vol.py linux_dump_map -p 10471 -D linux_dump_map_output/
```

Note that the `linux_dump_map` plugin extracts all pages that are accessible to the process, including pages that the process can access in kernel memory. The `strings` utility can be used to extract all strings from the memory regions to a new text file. To also extract Unicode strings, the `strings` utility should be run a second time with the option `-e1`.



```
$ strings -a linux_dump_map_output/*.vma > opera_browser.txt
$ strings -a -e1 linux_dump_map_output/*.vma >> opera_browser.txt
```

The resulting text file can then be searched with the `grep` utility. This technique enabled to find back the same HTTP Host headers that were found previously with the `linux_yarascan` plugin.

```
user@linux:~$ grep 'Host: ' opera_browser.txt
Host: ofa-r-sub.osp.opera.software
Host: www.earthrangers.com
Host: www.ecoheros.ca
```

This approach can also be used to extract text messages from memory. The following example shows text message content and metadata that was extracted from the Kik Messenger process.

```
user@linux:~$ grep '<message ' process_strings.txt
<message type="chat" to="bob430_934@talk.kik.com" id="a13e4a21-bc53-465a-92e3-86a3d23a2bf3" cts="1607497841972">
<body>Hi there</body><pb></pb><preview>Hi there</preview><kik push="true" qos="true" timestamp="1607497841972" /
<request xmlns="kik:message:receipt" r="true" d="true" /></ri></ri></message>
```

This data reveals the message content, recipient, timestamp and more. Each message was stored in a structured format: encapsulated by a message tag. By searching for all strings containing the text “<message ”, all memory-resident messages could be extracted from memory.

The extraction of text messages from memory could potentially reveal messages that were deleted from the flash memory, or that were stored only in an encrypted format on the flash memory. Some instant messaging applications such as WhatsApp may be configured to require additional fingerprint authentication. Inspecting the memory of the process could still reveal messages that might otherwise not be accessed on the live Android device.

Inspection of the memory image also revealed data structures that contained wireless access points information such as SSID and security key (PSK). This data resided in the wpa_supplicant process. Note that the Android device rebooted after it connected to the access point for the first time, so the PSK was automatically loaded into memory again after rebooting the device.

```
user@linux:~$ grep 'network={' Android_memory_strings.txt -A4
network={
  ssid="Network lab"
  psk="nfGe6qxJXs-QqAn7MU"
  key_mgmt=WPA-PSK
  priority=1
network={
  ssid="Guest network"
  key_mgmt=NONE
  priority=3
```

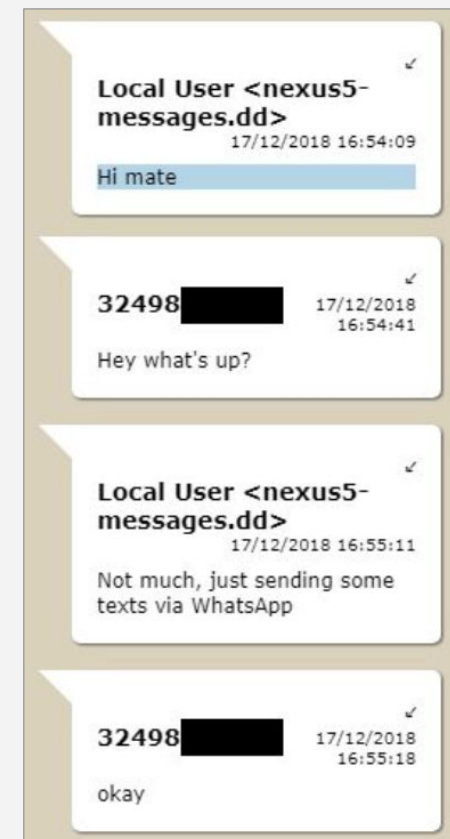
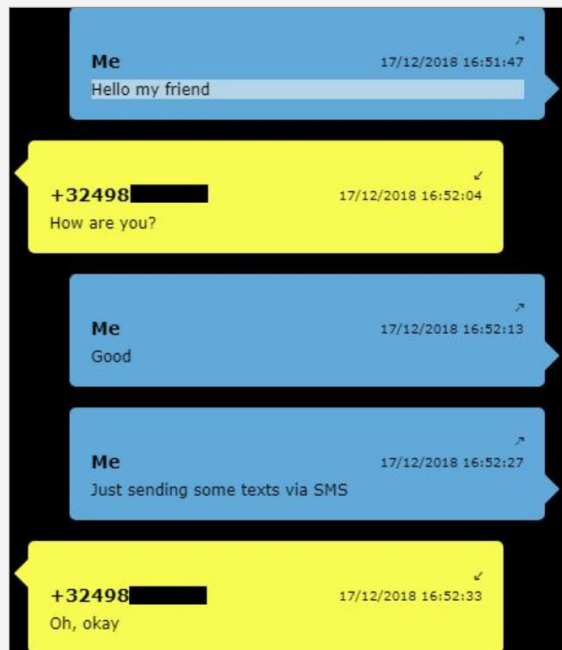
In some cases, it might be difficult for the analyst to know which signature to use to match relevant patterns in memory. For instance, searching for the PIN of a SIM card that was unlocked on the Android device is not feasible by simply matching all strings consisting of 4 digits, because there would be too many irrelevant hits. One possible approach is to introduce the same kind of artefact on a test device. For instance, when interested in how to obtain the PIN of a SIM card from memory, you can insert a SIM card with a known PIN in a test device,

unlock the SIM card with the PIN, take a memory image of the test device and then search for the known PIN in memory. This might reveal data in the near proximity of the PIN that can be used to create a signature that matches the PIN data on other devices as well. In the following example, looking for the known PIN “3961” revealed nearby data in the memory of the test device such as “supplyPin()”, “simSlot” and “pin:”, which could be used to create a signature.

```
user@linux:~$ grep 3961 Android_memory_strings.txt -C2
+++++xy WakeLock start : pid=2009, name=, type=0, tid=494, pid=483, pName=null
plugin
supplyPin() + simSlot:0 pin:3961
KeyguardViewMediator
setHidden false
```

Some digital forensics suites may be able to extract valuable system and user application artefacts from memory by using their built-in signatures, even if the suites are not designed to analyse memory images. The reason for this is that many forensic suites are designed to recover data from hard drives by using file carving and pattern matching techniques, in order to recover files and artefacts from unallocated space or from hard drives with a corrupt filesystem. Some suites can extract files and artefacts from memory images in the same way. For instance, the commercial forensic tool called Magnet Axiom could be used to extract valuable artefacts from a memory image of the Nexus 5 device.

The analysis of Android memory was not supported by the tool. However, by changing the file extension of the memory image to “.dd”, the tool was tricked into processing the memory image as a raw disk image. Of course, no file system was present in the image, but the tool analysed the data in the same way as it would analyse unallocated disk space, and was able to extract SMS messages, WhatsApp messages, Google search queries, Facebook activity and more. The following screenshot shows messages that were extracted by Magnet Axiom from the default Android Messages app (left) and from WhatsApp (right). Note that these are screenshots of the Magnet Axiom tool, which visualized the chat messages in a format that represents a messaging application interface. These are not actual screenshots of the Android device.

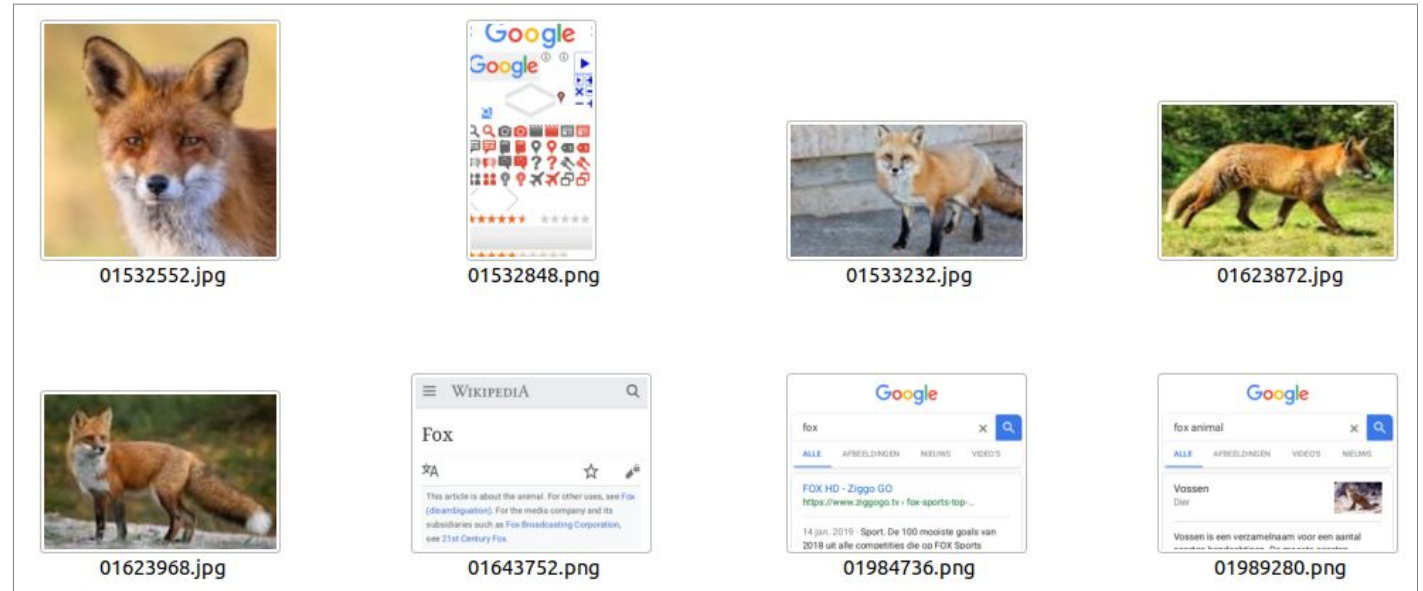


4.2.2. File carving

File carving techniques can be used to extract files from memory images that are not stored on the flash memory of Android devices. In the following example, the term “fox” was searched on Google Images within the Firefox browser on the Nexus 5 device. The device’s memory was then acquired and the `linux_dump_map` plugin was used to extract Firefox’ process memory. The extracted memory was then carved by Foremost, which is a file carving tool that can be installed via Aptitude.

```
$ cat linux_dump_map/*.vma |
foremost -t png,jpg
```

Foremost was able to extract all thumbnail images that were loaded on the Google Images results page. Additionally, Foremost extracted screenshots of the browser from memory. These screenshots are automatically taken by Firefox and displayed in the overview of opened tabs. The following screenshot shows a couple of the extracted images.



5. Conclusion

This article has discussed and demonstrated the complete capture of volatile memory from Android devices and covered several analysis techniques to extract system and user data from the memory image. By using the techniques demonstrated in this article, we've shown it is possible to extract artefacts that cannot be extracted from the internal flash storage of the Android device, either because the artefacts are never stored on the flash storage in the first place, or because they are encrypted before being stored. Some of these artefacts, including passwords and text messages, could be of high value during forensic investigations.

At this moment, there are several technical limitations that could hinder the memory acquisition in a real-case forensic investigation. The first major limitation is the need to have root privileges on the Android device in order to acquire its memory as demonstrated in this article. It is likely that the investigated device in a real-case scenario is not rooted yet. Even though it is in most cases possible to root the Android device, the device will most likely need to be rebooted in the process, which would wipe the original contents of memory. The second major limitation is the need for the Android device to enable loadable module support. The default kernel of some Android devices does not enable module loading, in which case it is not possible to acquire the memory with a kernel module as demonstrated in this article. It is possible to replace the kernel of the Android device with a modified version which does enable module loading, but this requires the device to be rebooted, which wipes the original contents of memory. Therefore, future work would be useful to develop a more feasible method to acquire memory from the Android device without having to reboot the device, for instance by extracting memory via JTAG without removing the battery of the device in the process.



References

Ligh, M. H., Case, A., Levy, J., & Walters, A. The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory. New York: Wiley; 2014.

Sylve J, et al., Acquisition and analysis of volatile memory from android devices, Digital Investigation (2012), doi:10.1016/j.diin.2011.10.003

Author

Dominique Pauwels
dominique.pauwels@pwc.com

Thank you



© 2021 PwC. All rights reserved. PwC refers to the PwC network and/or one or more of its member firms, each of which is a separate legal entity. Please see www.pwc.com/structure for further details.