Streamlining Extraction and Analysis of Android RAM Images

Simon Broenner, Hans Höfken and Marko Schuba
Department of Electrical Engineering and Information Technology,
Aachen University of Applied Sciences, Aachen, Germany

Keywords: Digital Forensics, Android, Smartphone, LiME, Memory, RAM Forensics.

Abstract: The Android operating system powers the majority of the world's mobile devices and has been becoming increasingly important in day-to-day digital forensics. Therefore, technicians and analysts are in need of reliable methods for extracting and analyzing memory images from live Android systems. This paper takes different existing, extraction methods and derives a universal, reproducible, reliably documented method for both extraction and analysis. In addition the VOLIX II front-end for the Volatility Framework is extended with additional functionality to make the analysis of Android memory images easier for technically non-adept users.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

Extraction of information from smartphones is becoming progressively more important as the ubiquity of mobile devices, such as smartphones and tablets, increases ever further. Law enforcement agencies, technicians and even end users are in need of reliable methods for extracting data from mobile devices when extraction of data from nonvolatile memory is not possible or insufficient. One possible source for data is the RAM (Random Access Memory) of such devices.

Possible scenarios for the analysis of RAM extracted from an Android device include typical law enforcement situations, such as the forensic analysis of devices seized during the course of investigations, as well as situations often encountered by IT support technicians, such as a first response analysis of a malware infestation. The potential applications are similar to those of live memory analysis for traditional desktop (Windows, Mac, or Linux) systems and are important due to the sheer amount of personal information stored on a smartphone or tablet.

Android is, in essence, a Linux-based system (Begun, 2011). It runs a Linux kernel, generally compiled for the mobile processors usually found in smartphones, tablets and other low-power devices. The lion's share of mobile processing for Android devices takes place on the ARM microprocessor architecture, which necessitates cross-compilation of any native code for use on these devices.

The common core with modern Linux distributions ensures the existence of software tools for extraction and analysis of Android memory images, such as the LiME (Linux Memory Extractor) kernel module (Sylve, 2015) and the Volatility Framework for analysis of memory images from multiple operating systems (Hale, 2013a).

This paper describes the leverage of the aforementioned tools LiME and Volatility resulting in a reproducible, reliable approach for the extraction and analysis of memory images from Android devices.

2 TECHNICAL CHALLENGES PRESENTED BY ANDROID

The Android operating system, while based on a Linux core, presents multiple new challenges for forensic analysis when compared to traditional desktop operating systems. The analysis of data stored in RAM on a target device can be very beneficial in addressing these problems.

2.1 Encryption of Non-volatile Storage Devices

The first of these challenges is FDE (Full Disk Encryption) as well as general encryption of storage

Broenner, S., Höfken, H. and Schuba, M.
Streamlining Extraction and Analysis of Android RAM Images.
DOI: 10.5220/0005652802550264
In *Proceedings of the 2nd International Conference on Information Systems Security and Privacy (ICISSP 2016)*, pages 255-264
ISBN: 978-989-758-167-0
Copyright © 2016 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

devices on Android systems (cf. Android Open Source Project). Starting with Android 4.0 ICS (Ice Cream Sandwich), Android has offered an option for complete encryption of the device's file system. Activating FDE is a matter of ticking a single checkbox in the Android security settings menu:

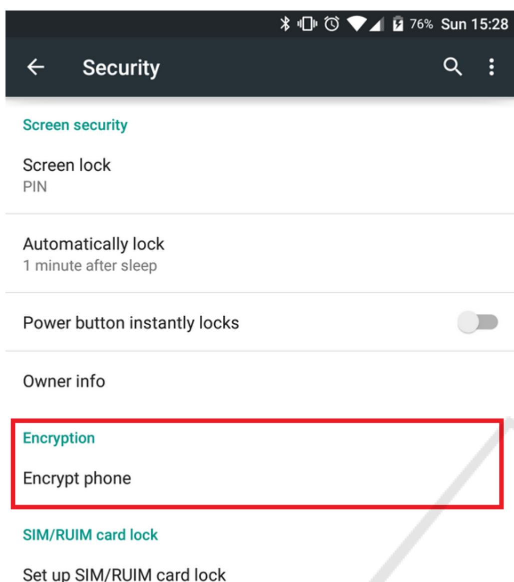

Figure 1: Activation of FDE on a smartphone running Android 5.0 Lollipop.

After the user has activated FDE, the device automatically encrypts internal storage devices using 128-bit AES (Advanced Encryption Standard) with cipher-block chaining (CBC) and an ESSIV (Encrypted salt-sector initialization vector) using SHA256. Starting from Android 5.0, this encryption can optionally be limited to the data partition in order to reduce performance penalties during the boot process (Android Open Source Project).

RAM, on the other hand, is generally not encrypted, as this is ill-advised from a performance standpoint. Therefore, if the data from an encrypted device's RAM were available, it would likely be possible to ascertain the encryption type and parameters of both standard and user-implemented encryption methods (such as LUKS). In some cases it may even be possible to circumvent encryption measures and decrypt the data on the device's encrypted volumes by obtaining the decryption key from within RAM.

Proofs of concept for the decryption of Android devices with storage encrypted by both dm-crypt (Elatov, 2015), the default device encryption solution used by Android starting from version 4.0, and LUKS have been published (Elatov, 2015; Müller and Spreitzenbarth, 2013). Both proofs of concept hinge upon the extraction of encryption parameters from a running device's RAM.

2.2 Locker/Vault Apps

The second challenge presented by Android is the ubiquity of locker/vault type apps (short: locker apps), which offer password-secured protection of sensitive data placed within them. These apps, ranging from simple and easy to circumvent to absolute state-of-the-art, are easily available from the Google Play Store or as downloadable APK files for manual installation (Begun, 2011) In addition, ADB and terminal emulators make it easy for advanced users to script their own security measures for sensitive data to be stored on the phone.

Similarly to attacks on FDE, analysis of RAM from devices running Android locker apps can provide valuable information as to the type and parameters of data protection methods used. In many cases, due to the simplicity of the methods used and an unhealthy reliance on security through obscurity, extracting data protected by locker apps is a simple matter of knowing where to look. The locker app's data in RAM often provides hints as to locations of files hidden by the app.

2.3 Android Malware

For support technicians and first responders, malware represents an additional problem in the Android ecosystem. Due to the sandboxing of Android applications (i.e., by the Dalvik VM used by Android for running its Java-based apps (Android Developers)) it is difficult to build malware scanning and protection apps which are able to analyze running code as well as any data apps may write to the device's internal storage.

Analysis of RAM from devices infected with malware can aid in detection, identification and even determining the origin of certain malware.

3 EXTRACTION OF MEMORY IMAGES

Multiple well-documented approaches for the extraction of memory images from Linux system exist (Caban, 2014). However, these approaches all present additional challenges when confronted with an Android system.

3.1 Android Kernel Version Fragmentation

Unlike desktop Linux distributions, which consist of known, largely homogenous quantities (i.e., large blocks of users of a single Linux distribution, such as Ubuntu LTS (Long Term Support) releases or Debian Stable), Android devices are very heterogeneous in terms of the kernel versions they run (Linux Profiles, 2012).

Android devices are generally updated OTA (Over The Air) by their manufacturers or software providers (these may include creators of custom ROMs, which are customized versions of the Android operating system for advanced end users). Any one of these updates can bring a minor kernel version change. Major kernel version changes, on the other hand, are generally reserved for major Android version updates, i.e., from Android 4.4 KitKat to Android 5.0 Lollipop.

This has led to large levels of fragmentation of the Android software ecosystem, due to nearly every device running its own custom compiled version of the Android kernel. Even two kernels with the same version number may not be alike due to custom compile flags used by the manufacturer to enable or disable functions and tweak the kernel towards the System on a Chip (SoC) and other hardware components in the device for performance and efficiency reasons.

3.2 Loadable Kernel Modules for Memory Extraction

Most Linux memory extraction strategies, including the approach presented here, are implemented through the use of LKMs (Loadable Kernel Modules (Caban, 2014)). The extraction of RAM from a Linux system requires compilation of an LKM specifically for the device in question. The LKM version number needs to match the kernel version number, the toolchain and compiler used to create the LKM need to be compatible with those used to compile the kernel, and the running kernel's source code is required for compilation of the LKM.

On desktop Linux systems, this is generally not an issue – one would simply install the same distribution, running the same kernel, on separate hardware, and then use that system to compile an LKM (Tilbury, 2013). Using the same distribution and kernel version as the target device makes compilation simple and generally rules out any compatibility issues. If a compromise in the level of forensic purity on the target system is acceptable, it is also possible to compile the LKM directly on the target system, ensuring compatibility. Tools which implement this for desktop systems automatically already exist – a prime example1 is Linux Memory Grabber by Hal Pomeranz (Hale, 2013b; Pomeranz, 2014).

Unfortunately, due to hardware constraints as well as a lack of software tools for the compilation process, this is not possible on Android devices.

Instead, when compiling an LKM for Android, compilation takes place on a separate system running a desktop Linux operating system. The compiler uses a cross compilation toolchain for CPU architecture compatibility (Hale, 2013a). Due to the availability of multiple toolchains and multiple compilers, the possibility of building incompatible LKMs exists. In general, using the toolchain and compiler used for compilation of the running device's kernel will likely lead to a working kernel module.

Multiple LKMs for the extraction of memory images are available, including fmem, pmem and LiME. In most cases, these LKMs work by creating a device such as */dev/fmem* as a parsable entry point for memory extraction (Hale, 2013b).

3.3 General LKM Compilation Process for Android Devices

Summarily, the compilation process consists of the following steps:

1. Initialization of the build environment: This step consists of the installation of prerequisites such as the required Java version and various libraries required for the Android build process, as well as the setting of required environment variables. ADB (Android Debug Bridge) access to devices via USB from within a Linux system also requires additional steps.
2. Procurement of a toolchain, such as the linux-arm-androideabi-toolchain included in the Android NDK.
3. Procurement of the target device's kernel source code.
4. Procurement of the LKM source code
5. Editing the Makefile included in the LKM source code to compile against the target device's kernel source code and use the correct toolchain
6. Issuing the *make* command

3.4 Linux Memory Extractor Lkm

LiME (Linux Memory Extractor) is a loadable kernel module used to extract memory from Linux-based systems. It has previously been used to successfully

extract the contents of RAM from Android devices and generates memory dumps compatible with the Volatility Framework (Sylve, 2011).

LiME was chosen as extraction tool due to the fact that it represents the most viable approach for Android devices. LiME is a well-documented project which is well-known to the developers of the Volatility Framework and hence provides a good starting point for the extraction of Android RAM images (Hale, 2013a).

During the course of this work, LiME was successfully used to extract memory dumps from both simulated and physical Android devices running Android versions up to 5.1 (Android Lollipop second release).

3.5 Extracting Memory with Lime

The use of LiME in order to extract memory from Android devices is much simpler than the compilation of the corresponding kernel module.

First, the LKM, in form of a .ko file, is copied to the device's internal memory or SD card. Then, from a terminal emulator app or an active ADB connection, the user executes the *insmod* command to inject the LKM into the running kernel and extract the device's memory to either a local file or over TCP. An example insmod command is the following:

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

Here the lime.ko LiME LKM file was placed on the device's internal memory (due to historical reasons, the mount point /sdcard/ is used for the user-accessible part of the internal memory on Android devices). The additional parameters *path* and *format* determine the location of the newly generated memory image and its structure, respectively (Sylve, 2015).

The process of dumping memory can take anywhere between a few seconds and multiple minutes, depending on the size of the device's RAM and the speed of the storage device being written to. After the memory dump has been generated, it can be transferred to the forensic workstation for analysis.

4 ANALYSIS OF PREVIOUSLY EXTRACTED MEMORY IMAGES

The analysis of memory images using Volatility requires a Volatility profile. The profile contains information about both the memory image as well as the system the image was extracted from. This includes information about the location of kernel debug symbols and the kernel's data structures (Hale, 2013b).

The Volatility Framework includes several precompiled profiles for popular operating systems (Linux Profiles, 2012) However, these are mostly limited to common major versions of Windows. The Volatility foundation also provides several optional profiles for Mac and Linux systems, available for download and manual installation. These optional profiles come in the form of a ZIP archive containing a precompiled Dwarfdump file, which contains the kernel data structure information, and the system.map file, which contains the kernel debug symbol information. The list of available profiles is small, including only a small number of the available Linux distributions and only a limited subset of each distribution's recent releases (Linux Profiles, 2012).

Precompiled profiles for Android devices may be listed in the future. However, the fragmentation of the Android software ecosystem will likely necessitate the custom compilation of a profile for each target device due to the sheer amount of possible combinations. The 10 most popular Android devices alone each run multiple different kernel versions over the course of multiple manufacturer updates, already offering up an incredible number of permutations.

4.1 Generating a Volatility Profile

Generating the Volatility profile requires the compilation of a module against the target device's kernel source to create the *vtypes* (kernel data structures). The Makefile included in the Volatility framework uses DwarfDump to pack the vtypes in a format which is readable by Volatility, generating a module.dwarf file (Hale, 2013b).

The resulting file is then packed in a ZIP archive together with the target device kernel's system.map file (Sylve, 2015). For Android devices the system.map is generally supplied along with the kernel source code, or can be manually assembled from information extracted from /proc/kallsyms/ using a *cat* command. The latter requires advanced knowledge of Linux kernel structures.

The following steps are necessary to generate a Volatility profile for an Android target device:

1. Procurement of the Volatility source code
2. Editing the included Makefile to use the target device's kernel source code as well as the correct toolchain for cross compilation
3. Issuing the *make* command

4. Procurement of the system.map file
5. Adding both the module.dwarf file generated by *make* and the system.map file to a ZIP archive

The newly created ZIP file can then either be placed in the volatility/plugins/overlays/linux directory of the Volatility package root directory or referenced via the –plugins flag during execution of Volatility from the command line (Raman, 2014).

4.2 Using Volatility to Analyze Android Memory Images

The Volatility framework is a simple command-line based tool. The plugins to be run are passed as parameters, i.e.:

```
python ~/android-volatility/vol.py
--profile=Android_Profile -f
~/lime.dump
linux_pslist >> results_pslist.txt
```

This command analyzes a memory image (*lime.dump*) using the linux_pslist plugin and outputs the generated list of running processes to a text file (*results_pslist.txt*).

Some plugins, such as *linux_volshell*, require additional parameters and input from the user.

5 THE VOLIX II VOLATILITY FRONTEND

In essence, the Volatility framework consists of a collection of Python scripts which are executed from the command line using a compatible version of the Python (a high-level object-oriented programming language with an on-the-fly interpreter) software package. The Volatility Foundation also provides standalone executable versions of Volatility for Windows, which are also run from the command line.

In an effort to simplify the operation of the Volatility framework, especially for less experienced users, the VOLIX Volatility frontend was developed (Logen et al., 2012). The latest version VOLIX II is built with a standard Windows Forms GUI, which should be familiar to most users of standard Windows software (VOLIX II, 2014).

As part of this work, the software of VOLIX II was extended to include functions pertaining to the analysis of Android memory images, as well as making the analysis of general Linux memory images simpler and more accessible to novice users. Changes include the addition of a Volatility profile selection mechanism for Linux and Android, additional Android-specific automated extraction routines for common tasks and an overhaul of the English language interface.

The user supplies the extracted memory image from the target device along with the Volatility profile. VOLIX II can then be used for execution of specific Volatility commands from within the GUI, or to execute automatic routines such as finding hidden processes or installed locker apps.

6 PRACTICAL CONSIDERATIONS FOR PHYSICAL ANDROID DEVICES

Due to the nature of the extraction methods presented here, target devices must fulfill certain prerequisites in order to be considered for successful extraction of an accurate memory image.

First and foremost, the injection of a kernel module such as LiME requires root access to the device. This means that unless the device has already been rooted for general use by its owner, the forensic investigator needs to find a viable root method and execute it on the device (Sylve, 2011).

Second, access to a console/command line is necessary for execution of the insmod command. There are two possible ways to access a command line: An ADB session or an app installed on the device (such as a terminal emulator or a remote command line app e.g. SSH server or similar).

Being able to open an ADB session requires either a USB or WiFi connection to the target device, the former of which is generally trivial unless there is physical damage to the device's USB port. However, in order to successfully connect to the device via ADB, the device must be set up to allow ADB connections in the developer options. In addition, recent Android versions have an ADB fingerprinting feature which requires the manual authorization of the connected PC on the target device itself.

Third, the injection of kernel modules via *insmod* requires that the kernel be configured to load modules at runtime. This requires three specific compilation flags (Hale, 2013b):

```
CONFIG_MODULES=y
CONFIG_MODULES_UNLOAD=y
CONFIG_MODULES_FORCE_UNLOAD=y
```

These flags are enabled by default for kernels running on many developer systems, as well as those included in many custom ROMs for Android devices.

This is not the norm, however, for factory standard devices sold to the general public.

Ideally, a target device would be previously rooted, have an *insmod*-friendly kernel configuration, and be unlocked for ADB connections from new PCs. In the real world, this is generally not the case. In order to root the device and allow ADB connections from a workstation, the forensic investigator must first gain access to the physical device, bypassing the device's PIN or pattern lock. Modifying the kernel to allow the injection of kernel modules generally requires a reboot, which may compromise system integrity (Ligh et al., 2014).

7 FORENSIC CONSIDERATIONS

Due to the nature of the extraction approach put forward here, a certain measure of forensic contamination of the target system is inevitable. Most devices encountered in the wild will not correspond to the ideal scenario introduced in Section 6, meaning they will need to be modified (root access, kernel modification, command line access) in order to extract memory images.

These modifications, in some cases, require a reboot of the system. This makes it difficult to preserve the exact memory state of the Android device for extraction and may lead to loss of useful data. Running applications and open files relevant to the investigation may not automatically run/open again after a reboot, and many caches and temporary files are purged during boot.

When applying the methods presented here, it is important to keep in mind that any extracted data may be incomplete and/or inaccurate.

In many cases, though, these shortcomings are irrelevant. Many Android applications start automatically at launch, as is the case for a lot of malware applications such as malicious RAT (Remote Administration Tool) apps.

8 EXTRACTION AND ANALYSIS OF MEMORY IMAGES FROM ANDROID DEVICES

As part of the research project memory images were extracted from multiple hardware devices including a Samsung Galaxy Nexus running Android 4.3 and 4.4, an AVD (Android Virtual Device) running Android 5.1, and a Google (LG) Nexus 5 running Android 5.1. While the Galaxy Nexus running Android 4.3 or 4.4 realistically represents a large portion of the currently available Android devices, extraction of a memory image from Android 5.1 running on Google's AVD emulator is largely a measure to ascertain that Android 5.1, as the most recent Android release at this time, contains no changes which would prevent the extraction method from working.

8.1 Android 5.1 Android Virtual Device

Building the LiME LKM, a Volatility profile and a compatible kernel for the standard version of Android 5.1 running on the AVD emulator is a straightforward affair. The kernel source code for the Android 5.1 version running on the emulator is easily obtainable directly from Google's code repository, and the Android system images included with the Android SDK (which contains the AVD emulator) are highly compatible with the arm-linux-androideabi-toolchains included in the Android NDK.

Cross-compilation of the LiME LKM required only small modifications to the Makefile included in the LiME source code, such as modification of the path to the Android NDK arm-linux-androideabi-toolchain and the path to the previously downloaded Android kernel source code.

Compilation of a kernel with the compilation flags needed for the use of *insmod* required the generation of a .config kernel configuration file using *make goldfish_armv7_defconfig*. The flags were then added to the kernel configuration file by hand before the kernel was compiled.

Generation of the Volatility profile for the Android 5.1 AVD was similarly simple, as was extraction using *insmod*, due to AVD system images providing root access by default.

After a successful extraction and transfer of the memory image to a forensic workstation via *adb pull*, the Volatility framework was successfully used to extract data from the image. Plugins used to extract data include *linux_pslist*, which provides a list of processes running at the time of extraction, *linux_dmesg*, which outputs the kernel message buffer, and *linux_lsof*, which lists open files at the time of extraction (cf. figure 2).

8.2 Samsung Galaxy Nexus

Both compilation of the LiME LKM and generation of the Volatility profile for Android 4.3 and 4.4 on the Galaxy Nexus proved more difficult.

8.2.1 CyanogenMod 11 (Android 4.4)

The first attempt to extract memory from this device was made using a nightly build of the CyanogenMod 11 custom ROM (based on Android 4.4), due to the fact that these systems present an easy target for the extraction methods detailed here. They offer enabled USB debugging (which means the device is ready for ADB connections), the ability to inject kernel modules and root access by default (see 6).

```
Offset     Name                 Pid             Uid
---------- -------------------- --------------- --------
0xd401cc00 init                 1               0
0xd401c800 kthreadd             2               0
0xd401c400 ksoftirqd/0          3               0
0xd401c000 kworker/0:0          4               0
...
0xd4345800 ext4-dio-unwrit      48              0
0xd4345400 jbd2/mtdblock1-      53              0
0xd4345000 ext4-dio-unwrit      54              0
0xd38cf400 jbd2/mtdblock2-      59              0
0xd38cf800 ext4-dio-unwrit      60              0
0xd42ac400 logd                 61              1036
0xd390bc00 healthd              62              0
0xd390b800 lmkd                 63              0
0xd390b400 servicemanager       64              1000
0xd390b000 vold                 65              0
0xd3913c00 surfaceflinger       66              1000
0xd3913400 qemud                68              0
0xd395c800 sh                   71              2000
0xd395c400 adbd                 72              0
0xd395c000 netd                 73              0
0xd399ac00 debuggerd            74              0
0xd399a800 rild                 75              1001
0xd399a400 drmserver            76              1019
0xd399a000 mediaserver          77              1013
0xd39cbc00 installd             78              1012
0xd39cb800 keystore             81              1017
```

Figure 2: Excerpt from the output of linux_pslist on an AVD Android 5.1 memory image.

Two problems were encountered during this approach: 1. The use of a nonstandard toolchain to build CyanogenMod and 2. the fact that CyanogenMod strips its kernel debug symbols to save space. The toolchain problem was solved by switching to a toolchain included in the CyanogenMod source code for full compatibility with the CyanogenMod custom ROM, enabling the successful extraction of a memory image from the Galaxy Nexus.

Obtaining the kernel debug symbols for the Volatility profile was possible only by using *cat* to extract a list of debug symbols from */proc/kallsyms* on the running system. After modification of the extracted list before integrating it in a Volatility profile, only rudimentary analysis of the extracted memory image proved possible.

Due to missing debug symbols, many important Volatility plugins, such as *linux_pslist* or *linux_dmesg* would output only empty files or a short string unrelated to the usual output of these plugins. Other plugins, such as *linux_check_syscall*, run without issue (cf. figure 3).

8.2.2 Galaxy Nexus Factory Image (Android 4.3)

In a more standard approach, the extraction and analysis method was also tested on the Galaxy Nexus running the standard Android 4.3 factory image. In this case, recompilation of the kernel with the compilation flags necessary for injection of kernel modules was required, but was otherwise straightforward.

```
Table Name      Index Address    Symbol
---------- ---------- ---------- ------------------------
32bit             0x0 0xc00bdc88 sys_restart_syscall
32bit             0x1 0xc00adf6c sys_exit
32bit             0x2 0xc0058e0c sys_fork_wrapper
32bit             0x3 0xc015936c sys_read
32bit             0x4 0xc01593e4 sys_write
32bit             0x5 0xc0158698 sys_open
32bit             0x6 0xc0156f7c sys_close
32bit             0x7 0xc00cb5d0 compat_sys_mq_timedsend
32bit             0x8 0xc01586d8 sys_creat
32bit             0x9 0xc0167a30 sys_link
32bit             0xa 0xc01677bc sys_unlink
32bit             0xb 0xc0058e1c sys_execve_wrapper
32bit             0xc 0xc0157f78 sys_chdir
32bit             0xd 0xc00cb5d0 compat_sys_mq_timedsend
32bit             0xe 0xc0167630 sys_mknod
32bit             0xf 0xc0158280 sys_chmod
32bit            0x10 0xc00e0e64 sys_lchown16
32bit            0x11 0xc00cb5d0 compat_sys_mq_timedsend
32bit            0x12 0xc00cb5d0 compat_sys_mq_timedsend
32bit            0x13 0xc0158c4c sys_lseek
32bit            0x14 0xc00b9df0 sys_getpid
32bit            0x15 0xc017786c sys_mount
32bit            0x16 0xc00cb5d0 compat_sys_mq_timedsend
32bit            0x17 0xc00e0f24 sys_setuid16
```

Figure 3: Excerpt from the output of linux_check_syscall on a Galaxy Nexus CyanogenMod 11 memory image.

Similar to the approach for the Google AVD emulator, the kernel source code for the Galaxy Nexus factory image is readily available, as the Galaxy Nexus was part of the Google Nexus developer device program. For this reason, the Galaxy Nexus factory image is also highly compatible with the toolchains supplied by Google's NDK. Cross-compilation of the LiME LKM as well as the module.dwarf file for the Volatility profile were completed without any problems.

Extraction of the memory image from the device additionally required gaining root access on the device, which was accomplished using the Nexus Root Toolkit.

After the successful extraction of the memory image and transfer to the forensic workstation via ADB, analysis of the image with Volatility proved successful, allowing the recovery of information with most of the Linux plugins included with Volatility.

8.3 Google (LG) Nexus 5

The Nexus 5 is part of Google's current Nexus lineup of developer smartphones (soon to be replaced by a

successor known as the Nexus 5 2015), and has, at time of publication, received an update to the latest Android release: Lollipop 5.1.1.

Like the Galaxy Nexus, it runs a standard factory image. Hence, cross-compilation of the LiME LKM as well as creation of the corresponding Volatility profile were straightforward and analogous to the process for the Google AVD emulator. Gaining root access was also possible using the same Nexus Root Toolkit used for the Galaxy Nexus.

After successful extraction of the memory image from the Nexus 5 device and transfer to the forensic workstation via *adb pull* over a USB connection, nearly all of the Volatility framework's Linux plugins were found to be useable, even with this very latest version of Android.

8.4 Simplifying Memory Extraction and Analysis

Successful extraction of a memory image using the LKM approach, as well as subsequent analysis using the Volatility Framework, depends on a number of factors which may vary according to the manufacturer, the version of the installed operating system, user settings, and the device's security and root status.

While the extraction and analysis of memory images from standard developer devices such as the Google Nexus series or the Google AVD Emulator follows a standard, documented routine, extracting and analyzing memory from any of the myriad available consumer devices requires a more specialized approach with time-consuming experimentation.

8.4.1 LKM and Volatility Profile Repository

At the lowest level, it is the generation of the LKM as well as the Volatility profile that differs from device to device. Since both the LKM and the Volatility profile only need to be generated once for a given device running a given Android version, a repository of kernel modules and ready-to-use Volatility profiles could greatly simplify the extraction of memory images from popular devices.

While the creation of a repository would require physical access to all the devices in question and, in some cases, multiple hours of work per device, crowdsourcing the modules and profiles is a possibility.

The Volatility Foundation already provides a basic set of Linux and Mac OS profiles for use with the Volatility Framework on Github at *https://github.com/volatilityfoundation/profiles*. This repository could be extended to include profiles for Android devices, preferably bundled with the corresponding LiME kernel modules. The kernel modules and profiles generated for the purposes of this research will be submitted shortly.

The availability of a LKM and Volatility profile for a given device would not guarantee a successful extraction – it would, however, greatly accelerate the process.

8.4.2 Different Versions of Android on the Same Device

While the generation of the LKM and the Volatility profile is a process that differs greatly from device to device, the process is generally very similar when the device is the same, but a different Android version is used.

In many cases, the only difference is the kernel source code that the LKM and Volatility profile vTypes are compiled against – simply replacing the source code directory with the appropriate version for the kernel now running on the device is often sufficient to generate a working LKM and Volatility profile.

In part, this is due to the significant changes manufacturers make to the kernels shipped on their devices (which generally do not change as heavily when the OS version is simply upgraded), but also simply due to differing kernel source code folder structures and incompatbile cross-compilation toolchains.

9 MALWARE ANALYSIS USING MEMORY IMAGES

The Linux plugins for the Volatility Framework provide multiple possible methods for the detection and analysis of malware in Android memory images. These range from the simple detection of a running process known to be associated with malware (using *linux_pslist*, for instance) to more complex methods such as using *linux_yarascan* for pattern matching malware detection (Luttgens et al., 2014), as well as ways to ascertain the addresses of remote communication servers if in use.

Well known Android malware such as the Remote Administration tool AndroRAT, for instance, was easy to detect running in memory even with the simplest detection routines, showing up in *linux_pslist* without any attempt at hiding itself (cf. figure 4)

While it is relatively simple for a malware developer to hide a malware application from the Android task switcher or DDMS (Dalvik Debug Monitor Service), hiding the application process so that it is not detectable in pslist is very difficult. Therefore, many malware apps rely on simple security by obscurity to avoid detection – such as the process name chosen by the developers of AndroRAT as seen in Figure 5.

```
0xc80a0800 externalstorage     799
0xc673ac00 d.process.media     865
0xcca5d800 putmethod.latin     880
0xc805ec00 ndroid.keychain     903
0xccb38000 .android.dialer     930
0xc829ac00 viders.calendar     947
0xc827dc00 gedprovisioning     966
0xc8343c00 com.android.mms     985
0xccb84800 ndroid.settings     1015
0xc6429c00 ndroid.calendar     1032
0xc8146800 droid.deskclock     1054
0xc8146400 m.android.email     1073
0xc43cd400 ndroid.exchange     1089
0xd1e67c00 my.app.client       1115
0xc2126c00 m.android.music     1134
0xc3ec5400 sh                  1297
0xcefb7800 kworker/0:1         1305
0xc3ec5800 flush-179:0         1306
0xcefb7000 insmod              1307
```

Figure 4: Excerpt from the output of linux_pslist from a system with a running AndroRAT installation – the process name chosen by the developers of AndroRAT is very generic and could correspond to any number of applications a user has installed.

After detection, it may be useful to ascertain the servers and/or nodes the malware communicated with – this can be achieved by reading the data in the routing table cache using *linux_route_cache* (Pryor, 2013). The routing table cache can contain a record of systems a Linux machine has communicated with in the past:

If the malware was transmitting at the time the memory image was extracted, it may also be useful to extract packets stored in the send and receive queues

```
Volatility Foundation Volatility Framework 2.4
Interface        Destination          Gateway
---------------- -------------------- -------
wlan0            8.8.8.8              192.168.0.1
lo               255.255.255.255      255.255.255.255
lo               255.255.255.255      255.255.255.255
wlan0            157.56.53.46         192.168.0.1
lo               255.255.255.255      255.255.255.255
lo               255.255.255.255      255.255.255.255
wlan0            104.40.141.105       192.168.0.1
lo               255.255.255.255      255.255.255.255
wlan0            111.221.74.32        192.168.0.1
wlan0            134.170.107.176      192.168.0.1
lo               192.168.0.43         192.168.0.43
wlan0            1.2.3.4              192.168.0.1
lo               192.168.0.43         192.168.0.43
lo               255.255.255.255      255.255.255.255
lo               192.168.0.43         192.168.0.43
```

Figure 5: Excerpt from the output of linux_route_cache from a Google Nexus 5 running Android 5.1.1 - note the destination IP addresses accessed through gateways 192.168.0.1 and 192.168.0.43 for access via WiFi and mobile data respectively.

at the time of extraction. This can be achieved using *linux_pkt_queues*, which recovers packets from the queues and can save them directly to disk (Pryor, 2013).

Additionally, *linux_sk_buff_cache* can be used to recover and save packets which were in kernel memory at the time of extraction, but is currently incompatible with SLUB (Unqueued Slab Allocator – a memory management mechanism which allows the retention of certain allocated data objects for later reuse (Cinar, 2015)), the current default allocator in Linux since kernel 2.6.23 (Case, 2012).

Last but not least, the Volatility framework also provides Linux plugins which allow the detection of active rootkits, such as *linux_check_creds*, which can detect rootkits piggypacking on the credentials of processes with root privileges (often PID 1), *linux_check_idt*, which lists the addresses and symbols contained within the IDT (Interrupt Descriptor Table), including those hooked by rootkits, and *linux_check_syscall*, which lists the system call tables and checks them for functions hooked by rootkits (Linux Profiles, 2012).

Rootkits may also inject kernel modules which are hidden in the loaded module list (as called by *lsmod*). The Volatility plugin *linux_check_modules* can detect these modules, as long as they are still present under */sys/modules/* (Pryor, 2013).

10 CONCLUSION

The extraction and analysis of RAM images from Android devices remains a relatively young field. While possibilities for both extraction and analysis exist for a multitude of Android device and software constellations, the application of the currently known and documented methods generally require extensive knowledge of both Linux systems and the use of *Make* for the compilation of kernel modules. In addition, many of the existing approaches are documented only for specific devices running specific versions of Android, making it difficult to apply them to other devices which a forensic investigator may encounter during investigations.

The methods described here provide a universal approach for the extraction of RAM images from Android devices, as well as the generation of a corresponding Volatility profile for subsequent analysis using the Volatility framework. In addition, the extension of the VOLIX II Volatility frontend for Android memory images greatly simplifies the analysis of these memory images with the previously generated Volatility profiles.

In the future, automatic (scripted) generation of LiME LKMs and Volatility profiles for a given device would greatly simplify the process of memory extraction and analysis. This could, for instance, be achieved via distribution of a Linux VM (virtual machine) containing the required scripts, to be launched on the investigator's forensic workstation.

With a target device that fulfills the requirements set forth in section 6, the investigator would need to supply only the source code for the kernel running on the device and quickly be able to compile a working LiME LKM for the device, as well as generate a Volatility profile for memory extracted from the device with the LKM.

A cleaner forensic approach for memory extraction from Android devices, free from the limitations shown in section 7, would require a completely new approach which does not hinge upon the injection of a kernel module. This would, however, require the discovery of a vulnerability in Android which can be triggered without modification of the running system. Even if such a vulnerability were located, it would only be a matter of time until Android was patched to remove the vulnerability.

REFERENCES

Android Developers, *Security Tips,* [Online], Available: http://developer.android.com/training/articles/security-tips.html [4 Sep 2015].

Android Open Source Project, *Encryption,* [Online], Available: https://source.android.com/devices/tech/security/encryption/index.html [4 Sep 2015].

Begun, D., A., 2011, *Amazing Android Apps for Dummies*, Wiley & Sons.

Caban, D., 2014, *Acquiring Linux Memory from a Server Far Far Away*, [Online], Available: http://blog.opensecurityresearch.com/2014/05/acquiring-linux-memory-from-server-far.html [4 Sep 2015].

Case, A., 2012, *Phalanx 2 Revealed: Using Volatility to Analyze an Advanced Linux Rootkit*, [Online], Available: http://volatility-labs.blogspot.de/2012/10/ phalanx-2-revealed-using-volatility-to.html [4 Sep 2015].

Cinar, O., 2015, *Android quick APIs reference*, Apress.

Elatov, K., 2015, *Recover LUKS Password from Android Phone,* [Online], Available: http://elatov.github.io/2015/03/recover-luks-password-from-android-phone/ [4 Sep 2015].

Hale, M., 2013a*, AndroidMemoryForensics - Instructions on how access and use the Android support*, [Online], Available: https://code.google.com/p/volatility/wiki/AndroidMemoryForensics [4 Sep 2015].

Hale, M., 2013b, *LinuxMemoryForensics - Instructions on how to access and use the Linux support*, [Online], Available: https://code.google.com/p/volatility/wiki/LinuxMemoryForensics [4 Sep 2015].

Ligh, M., H., Case, A., Levy J., Walters, A., 2014, *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*, Wiley.

Linux Profiles, 2012, *LinuxProfiles - Linux Profile Reference*, [Online], Available: https://code.google.com/p/volatility/wiki/LinuxProfiles [4 Sep 2015].

Logen, S., Höfken, H., Schuba, M., 2012, *Simplifying RAM Forensics - A GUI and Extensions for the Volatility Framework,* Proceedings of 5th International Workshop on Digital Forensics, Prague, Czech Republic.

Luttgens, J., T., Pepe, M., Mandia, K, 2014, *Incident Response & Computer Forensics*, 3rd edition, McGraw-Hill Education.

Müller, T., Spreitzenbarth, M., 2013, *FROST – Forensic Recovery of Scrambled Telephones,* in Applied Cryptography and Network Security, 2013, Eds. Jacobson, M., Locasto, M., Mohassel, P., Safavi-Naini, R., Springer.

Pomeranz, H., 2014, *Linux Memory Grabber - A script for dumping Linux memory and creating Volatility(TM) profiles*, [Online], Available: https://github.com/halpomeranz/lmg/blob/master/README [4 Sep 2015].

Pryor, K., 2013, *Volatility Linux Profiles*, [Online], Available: http://digiforensics.blogspot.de/2013/12/volatility-linux-profiles.html [4 Sep 2015].

Raman, S., 2014, *Installing Linux Profile in Volatility*, [Online], Available: https://shankaraman.wordpress.com/2014/05/23/installing-linux-profile-in-volatility/ [4 Sep 2015].

Sylve, J., T., 2011, *Android Memory Capture and Applications for Security and Privacy*, M.S. Thesis, University of New Orleans, New Orleans.

Sylve, J., T, 2015, *LiME ~ Linux Memory Extractor*, [Online], Available: https://github.com/504ensicsLabs/LiME/blob/master/README.md [4 Sep 2015].

Tilbury, C., 2013, *Getting Started with Linux Memory Forensics*, [Online], Available: http://forensicmethods.com/linux-memory-forensics [4 Sep 2015].

VOLIX II, 2014, *Volatility Interface and Extensions,* [Online], Available: http://www.it-forensik.fh-aachen.de/projekte/volixe, [4 Sep 2015].