Retrofitting Mobile Devices for Capturing Memory-Resident Malware Based on System Side-Effects

Zachary Grimmett, Jason Staggs, Sujeet Shenoi

▶ To cite this version:

Zachary Grimmett, Jason Staggs, Sujeet Shenoi. Retrofitting Mobile Devices for Capturing Memory-Resident Malware Based on System Side-Effects. 15th IFIP International Conference on Digital Forensics (DigitalForensics), Jan 2019, Orlando, FL, United States. pp.59-72, 10.1007/978-3-030-28752-8_4 . hal-02534602

HAL Id: hal-02534602
https://inria.hal.science/hal-02534602

Submitted on 7 Apr 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Chapter 4

RETROFITTING MOBILE DEVICES FOR CAPTURING MEMORY-RESIDENT MALWARE BASED ON SYSTEM SIDE-EFFECTS

Zachary Grimmett, Jason Staggs and Sujeet Shenoi

Abstract Sophisticated memory-resident malware that target mobile phone platforms can be extremely difficult to detect and capture. However, triggering volatile memory captures based on observable system side-effects exhibited by malware can harvest live memory that contains memory-resident malware. This chapter describes a novel approach for capturing memory-resident malware on an Android device for future analysis. The approach is demonstrated by making modifications to the Android `debuggerd` daemon to capture memory while a vulnerable process is being exploited on a Google Nexus 5 phone. The implementation employs an external hardware device to store a memory capture after successful exfiltration from the compromised mobile device.

Keywords: Mobile device malware, system side-effects, memory capture

1. Introduction

Mobile devices are increasingly being used to process and manage personal and sensitive information such as photos, videos, browsing history, notes, social media posts and bank account data. As a result, these devices have become attractive targets for adversaries and attacks on the devices are increasing in their scope and magnitude [5].

Mobile devices share several attack vectors with traditional workstations (e.g., Wi-Fi and Bluetooth adapters). However, mobile devices are also continuously connected to cellular networks in which the device owners have little to no control. Fragmentation in mobile device operating systems and embedded device architectures makes it difficult to

develop exploits that impact multiple devices, but it also renders the mobile device ecosystem more challenging to secure.

Vulnerabilities that affect large families of devices have been demonstrated [1, 3, 4, 12]. These vulnerabilities make it imperative that new efforts be developed to secure mobile devices against increasingly sophisticated attacks. Analyzing and understanding the rapidly evolving threats to mobile devices require the capture and analysis of evidence pertaining to attacks on the devices.

Memory-resident malware is difficult to detect because it resides entirely in volatile memory and does not write to secondary memory. Additionally, this type of malware often removes itself from memory after execution. This makes it impossible for a forensic analyst to identify and collect malware after a compromise has occurred. The only option is to take proactive measures to capture the contents of memory while the malware still resides in memory. In addition to supporting forensic investigations, the ability to capture the malware enables researchers to identify and mitigate the vulnerabilities exploited by the malware.

Mobile devices are exposed to unique threats compared with stationary devices (e.g., workstations) because of their mobility. Moreover, real-world mobile devices incorporate peripherals such as communications processors that are not present in most virtual or emulated devices. Therefore, it is important to leverage real-world mobile devices to understand and mitigate the unique threats.

The proposed approach leverages digital forensic and embedded device engineering techniques to capture evidence of malicious activity on mobile devices [8]. Consumer hardware, specifically a Google Nexus 5 smartphone, was adapted to capture transient malware, and multiple techniques for storing the captured information are evaluated. The Stagefright family of exploits is used as a case study to explore and identify strategies for detecting various types of malware.

2. Malware Categorization

Security monitoring solutions typically rely on identifying malware based on artifacts (e.g., data or code) that reside in a filesystem and/or by examining how malware behaves during execution [2, 7]. Malware is identified by developing and checking for storage signatures corresponding to malware artifacts and/or execution signatures that describe malware behavior. Malware developers attempt to elude signature-based detection by making slight modifications to malware code and/or behavior. In turn, malware analysts attempt to generalize the storage and execution signatures to detect variations of the same malware. Although

these approaches may work to varying degrees for known malware, they cannot be applied effectively to (unknown) malware that has not been studied previously.

Grimmett et al. [6] have proposed alternative methods for identifying malware based on observable system side-effects. They also present a taxonomy for categorizing malware according to its behavior and system side-effects. The taxonomy covers three categories of malware based on: (i) user-detectable behavior; (ii) system-detectable behavior; and (iii) inconspicuous behavior. Each malware category exhibits different characteristics that can be leveraged to develop system side-effect signatures for detecting and capturing the malware in question.

Grimmett et al. [6] also present a case study involving the Stagefright malware. Stagefright is designated as system-detectable malware because it produces side-effects that are detectable by the underlying operating system (i.e., Android). The system side-effects are a result of repeated attempts at exploiting a system service that causes a service to crash (i.e., brute force execution). Due to the reliability requirement imposed on mobile devices, critical services automatically restart after a crash (e.g., due to a failed exploit attempt), enabling an attacker to attempt to exploit the vulnerability in the system service again. In some instances, the crashed system service is transparent to the end-user; this enables an attacker to attempt the exploit repeatedly until it succeeds and without alerting the user.

The crashing of a service as a result of a failed exploit attempt is, in fact, a side-effect that is observable to the underlying operating system. As a result, this side-effect can be used to trigger events that could assist a malware analyst in identifying and collecting previously unknown malware.

2.1 Stagefright

The Stagefright family of vulnerabilities include integer overflows and heap overflows deep in the MPEG4 media processing portions of the `libstagefright` Android operating system library [12]. The vulnerabilities are critical because they can be triggered remotely by sending specially-crafted MMS messages to mobile device users. The entire exploitation process is transparent to a user in that it does not trigger warnings or error messages.

In order for a Stagefright exploit to succeed, it has to defeat address space layout randomization (ASLR). Address space layout randomization is a memory protection mechanism supported by most modern operating systems to mitigate memory corruption exploitation attempts.

Address space layout randomization attempts to randomize the base addresses of key components of a process (e.g., libraries, the stack and the heap) to make it more difficult for an attacker to reliably jump to a known piece of code in memory.

To overcome this barrier, Stagefright guesses the locations of the base addresses of the `libstagefright` library in the `mediaserver` process. When the locations are guessed incorrectly, the Android `mediaserver` process simply crashes and restarts, reloading the process into the same vulnerable state. Thus, multiple exploitation attempts and subsequent crashes tend to occur when a Stagefright exploit is executed. Since the `mediaserver` process runs at a privileged level, successful exploitation of the `libstagefright` library in the `mediaserver` process enables the attacker to inherit system-level permissions. These characteristics make Stagefright an excellent candidate for demonstrating that system side-effects produced by malware can be used to capture the malware while it is still in volatile memory. As a result, the Stagefright family of vulnerabilities is considered as a case study in this research.

2.2 Live Memory Analysis

A number of tools have been developed for acquiring memory images from volatile memory (i.e., RAM) [9–11]. A widely used open-source tool is `Linux Memory Extractor` (`LiME`). `LiME` is a loadable Linux kernel module that can dump the entire physical memory of a device. In an attempt to be forensically sound, `LiME` is designed to have a very limited memory footprint. These characteristics make `LiME` a useful tool for collecting evidence of malicious activity that cannot be precisely located in memory.

Certain challenges must be addressed in order to use `LiME` to capture malware. First, `LiME` has minimal impact on the target system. Since `LiME` does not halt the system, it is necessary to ensure that the downloaded malware remains in memory when the capture process executes. Second, the memory image produced by `LiME` is the size of the device physical memory – this is about 2 GB in the case of a Google Nexus 5 device. The memory image file can be stored on device (local) storage or saved over a TCP connection to a remote machine. When network access is not available, the number of captures that can be stored are limited by the amount of storage space available on a mobile device.

While `LiME` is ideal for collecting large amounts of memory at a given time, it is not the best choice for consistent or continuous monitoring of live memory. This makes `LiME` useful in situations where it can be invoked when suspicious activity such as a system-detectable side-effect is

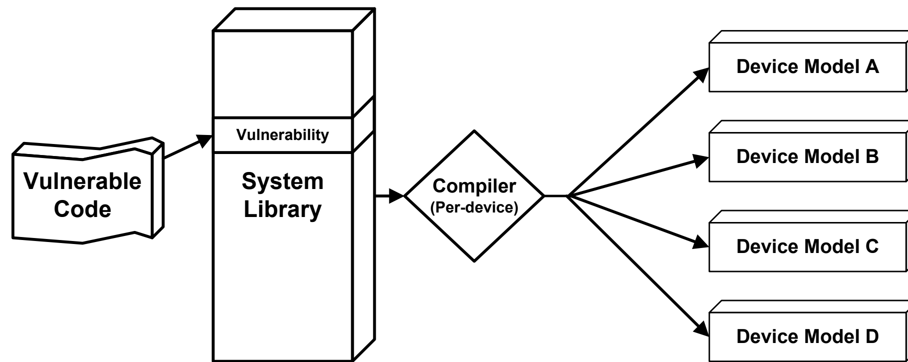

Figure 1. Proliferation of a vulnerability in a system library.

detected. The case study described in this chapter focuses on capturing malware from mobile devices. However, the captured malware is only useful if analysts can examine and understand what has been captured.

3. Automated Memory Acquisition

The mobile device malware analysis community lacks a mechanism for reliably and feasibly capturing a snapshot of memory during system exploitation attempts. This section describes a proof-of-concept implementation that demonstrates the viability of automated memory acquisition from an Android mobile device. The proof-of-concept has been implemented on a Google Nexus 5 phone. By modifying the Android `debuggerd` daemon, the physical memory contents are dumped upon invoking the `LiME` kernel module during the crash of a system process. Because of the limited storage on the mobile device, the memory capture is subsequently exfiltrated to another device using TCP via USB forwarding.

3.1 Design Requirements

This research was motivated by the concern that a vulnerability in a system library (e.g., Stagefright) puts a large number of mobile devices at risk for remote exploitation [4, 12]. The focus on system library vulnerabilities is important. This is because, to maximize the impact, malware developers invest resources in identifying vulnerabilities and exploits in the system libraries of popular operating systems.

Figure 1 demonstrates how a vulnerability in a system library becomes a vulnerability for every device that uses the library. Additionally, since services traditionally execute with higher privileges than user applications, attackers have an additional incentive to exploit the services.

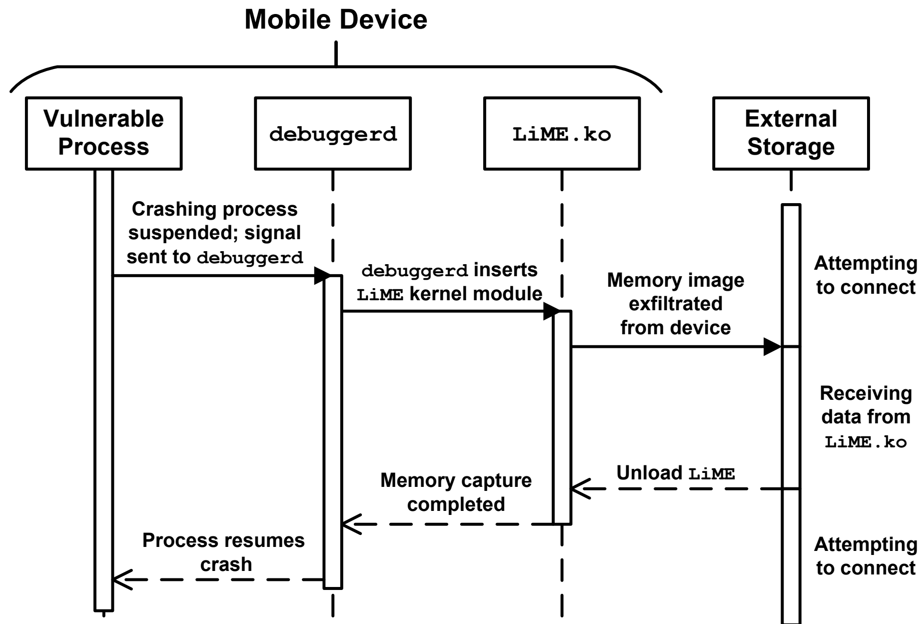

Figure 2. Malware capture process.

The software modifications that support live memory acquisition from a mobile device should be reliable and should have minimal impact on observable device behavior. The modifications must also run on a physical device so that all the potential vectors are available for study. The hardware should require as few proprietary modifications to the operating system as possible. Additionally, the modifications should be adaptable to newer devices and future operating system versions in order to meet future malware analysis needs.

3.2 Implementation and Testing

Figure 2 presents the process for acquiring a memory image from a mobile device after a service has crashed. When a service crashes, the `debuggerd` handler is signaled, which suspends the crashing service. The `debuggerd` daemon also initializes the `LiME` kernel module and specifies the capture parameters (e.g., image format and exfiltration method). The primary use case for this implementation is to transfer the image using TCP via USB forwarding; however, transfer via Wi-Fi is also supported by the implementation. Additionally, the acquired memory image may be moved to device local storage for transfer at a later time.

The unmodified `debuggerd` daemon suspends all crashing processes when it generates a tombstone file – it is after `debuggerd` has completed

its own crash handling functionality that the process is left suspended and gdb is attached or the process is allowed to continue and crash. Because debuggerd suspends a process while gathering information, additional code added to debuggerd can execute before the process is allowed to resume. After the image acquisition is complete, the kernel module is unloaded and debuggerd allows the process to resume and crash. The malware capture hardware stores the acquired image and proceeds to wait for another memory acquisition; the crashed process may then be restarted by the operating system.

3.3 Android Modification Results

A framework was created to manage a (mobile) device-under-test and enable automated testing. The framework, which was developed in Python, creates test instances that use the Android Debug Bridge (adb) to interface with Android devices. These test objects can be extended to create new test objects with additional functionality as desired. The tests were extended to enable automated testing of the LiME kernel module and to verify that the acquired memory samples contained the target crash vectors.

The crash vector was recovered from memory in order to determine if the entire vector was captured successfully. The Volatility plugin linux_pslist was used to determine the mediaserver process identifier. Next, linux_yarascan was used to search the virtual memory of the process for ftyp, which denotes the "File Type Box" that appears in the beginning of some MPEG4 media files (e.g., crash vector). Next, linux_proc_maps was used to determine the mapped memory sections that needed to be extracted for analysis, upon which linux_dump_map created an image of the relevant memory mapping from the mediaserver process. A Python script was written to verify that the dumped memory contained the crash vector.

The LiME kernel module is designed to provide minimally-invasive memory acquisition for forensic analysts. The proof-of-concept implementation does not assume that the device is handled using digital forensic best practices. Therefore, additional testing had to be conducted to verify that data remains in memory long enough to be captured using LiME. Additionally, any differences between suspended and non-suspended processes had to be understood to determine the impact of suspension on memory acquisition.

To determine if LiME was suitable for the proposed tasks, tests were conducted to measure how effectively LiME captures an MPEG4 crash vector when it is executed manually immediately following a browser

Table 1. Crash vector capture success rates during manual testing.

	No Wait after Reboot		**Wait after Reboot**	
	Local Storage	**TCP Capture**	**Local Storage**	**TCP Capture**
Suspended	25%	0%	100%	100%
Not Suspended	0%	50%	0%	100%

crash. The tests were performed using a capture to local device storage and exfiltration via TCP over a USB connection to an external host. Additionally, tests were performed with and without processes set to suspend and wait after a crash. Moreover, the tests were executed with and without a one-minute wait between restarting the device and performing the crash and memory acquisition.

The results in Table 1 demonstrate the impact of a short wait on the success rate. The wait/no-wait results demonstrate that the target data is likely to be lost unless it is captured quickly or the process is suspended. In the experiment, the device-under-test was restarted before every memory acquisition test to ensure that no residual data remained from previous tests. During the Android startup process, the user interface was made available as quickly as possible and other startup tasks were executed in the background. Because the background startup operations were still initializing the system, memory was released and reused more rapidly than under normal operating conditions. Therefore, the startup period had to be allowed to complete or the memory acquisition would likely be disrupted by the high memory turnover.

Figure 3 shows the methodology for testing the reliability of capture of an MPEG4 crash vector using the proof-of-concept implementation. The device-under-test navigates to the crash vector (`crash.mp4`) on a local webserver, which causes `mediaserver` to crash. When `mediaserver` crashes, `debuggerd` handles the crash and inserts the `LiME` kernel module and performs the memory acquisition. The acquired memory image is then searched for instances of the known crash vector.

A successful capture includes at least one complete and intact instance of the crash vector. If the complete vector cannot be found in memory, it may be possible to find partial instances. The partial instances would be less valuable than a complete sample from the perspective of malware analysis. However, further investigation may enable a complete instance to be reconstructed from memory.

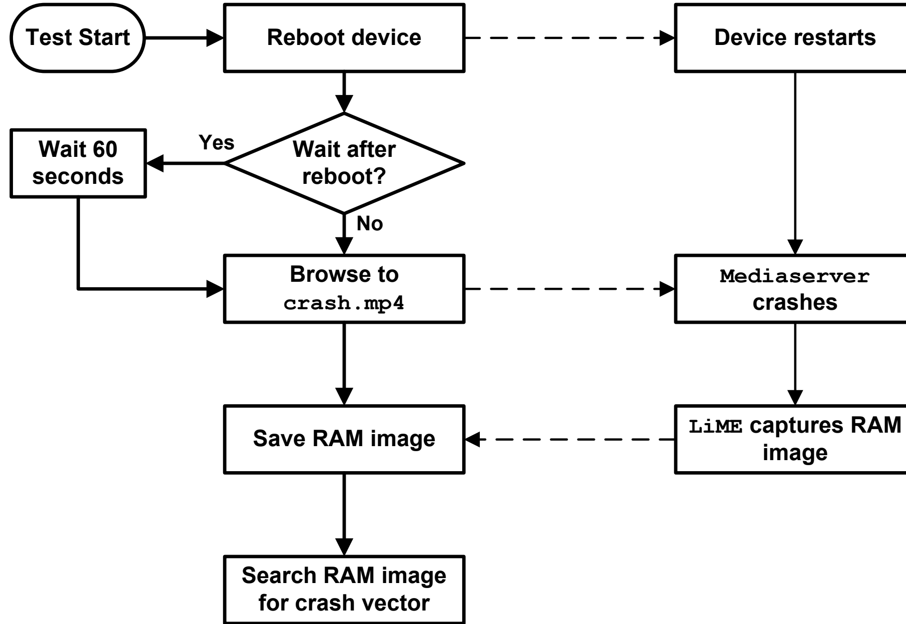

Figure 3. Test design for capture technique validation.

When `LiME` acquires memory and exfiltrates the image to a remote host via TCP, the rate of capture is limited by the network bandwidth between the device and remote host. The proof-of-concept implementation was designed to support capture via a Wi-Fi network. However, as discussed below, the time required to complete the capture limits the effectiveness of this approach.

Table 2. Average capture time for the exfiltration methods.

Exfiltration Method	Time (seconds)
Local Storage (Capture Only)	59.79
Local Storage and USB Downloading	446.95
TCP Exfiltration via USB Forwarding	382.22
TCP Exfiltration via Wi-Fi Network	2,382.93

Table 2 shows the average time required for various memory acquisition techniques. Note that the testing framework incorporates approximately 5% additional overhead for network operations using Python sockets, which is not enough to disrupt the experimental results. The significant increase in time required to acquire an image over Wi-Fi mo-

tivated the development of the portable hardware solution presented in the next section.

4. Hardware Enhancements

The amount of storage available on a Google Nexus 5 (and most phones for that matter) is limited and is certainly not ideal for storing multiple instances of full memory captures. This section describes a portable USB host solution that provides external storage capabilities for the memory capture system.

4.1 Design Process and Requirements

The memory acquisition proof-of-concept implementation described in the previous section leverages a connected USB host to capture an image using TCP over USB or to download a locally-stored memory capture. The implementation could be altered to support the local storage of multiple memory captures. However, a Google Nexus 5 device has just 16 GB internal flash memory and only 12 GB of this memory is free after installing the modified version of Android and the `Open GApps` package that contains Google Chrome. Additionally, a Google Nexus 5 does not support any removable storage (e.g., microSD card). Because each captured memory image is 2 GB in size, the number of captures that can be stored at one time is severely limited.

An external storage solution also reduces the likelihood of a captured memory image being erased or corrupted. Because this research has focused on malware capture for future analysis, it would not be prudent to rely on a compromised mobile device to preserve the captured image. Furthermore, since a USB connection to a compromised device could put the USB host at risk, the external storage solution should be easily wiped and redeployed as necessary.

It is also important that the hardware support package be portable. This requires the external storage solution to incorporate a battery, which imposes a limit on the length of time the hardware package can be used between charges. Thus, the portable hardware should consume as little power as possible while remaining reliable and available.

4.2 Implementation Details

Figure 4 shows the proof-of-concept implementation created using a Raspberry Pi 3 as a USB host. The Raspberry Pi 3 runs the Raspbian Jessie Lite operating system (a minimal operating system based on Debian Jessie) and includes `adb` binaries compiled with the same toolchain used by the software modification proof-of-concept system de-

Figure 4. Memory capture support hardware.

scribed above. The Raspberry Pi has an 802.11n wireless radio and multiple USB ports that enable it to support memory captures over Wi-Fi or USB.

The Raspberry Pi 3 is a general-purpose computing device that uses more power than a dedicated microcontroller. However, the availability of a Linux operating system enables the memory capture device to be more adaptable than an embedded device. The increased power requirement is a reasonable trade-off for the additional functionality and ease-of-use provided by the operating system. Specifically, the operating system enables the memory capture device to incorporate logic that controls the behavior of the capture software and determine when to download a completed local capture. Additionally, the device hosts an SSH server that enables the device to be remotely operated and configured.

The only way to safely shut down the Raspberry Pi is via the `shutdown` or `halt` commands – disconnecting the device from power without shutting it down properly could corrupt the microSD card and the captured memory images it contains. The implementation incorporates an Anker Power Bank with 8,400 mAh capacity and an external charge indicator. The external charge indicator should be monitored to minimize the risk of draining the battery and corrupting the captured memory images.

An alternative solution is to use a second mobile device to support the memory capture device. Using an Android device would eliminate the need for an external battery while enabling similar capabilities as a Raspberry Pi (i.e., Linux operating system and `adb` support). However,

Table 3. Average download times of memory images via `adb`.

ADB Host	Time (seconds)
MacBook Pro (2 GHz Intel Core i7)	317.40
Raspberry Pi 3 (1.2 GHz Cortex-A53)	422.50

the choice of mobile device is limited by the same constraints that motivate the use of a support device – that is, the device would need to provide substantial external storage. In any case, a Raspberry Pi costs less than any similar mobile device.

4.3 Experimental Results

The Raspberry Pi 3 has a less powerful processor than the workstations used to test the proof-of-concept memory capture implementation. As a result, the portable storage solution requires more time to perform the tasks than the times listed in Table 2. Table 3 presents the times required to download locally-stored memory images via `adb`.

Device power usage was measured using a USB power monitor between the battery and Raspberry Pi. The power monitor measured the total power consumed by the device and provided instantaneous current and power measurements.

Table 4. Power consumption of the support hardware.

Device Status	Power (Amps)
Device idle; no connection	0.26
Device idle; phone connected with screen off	0.35
Device idle; phone connected with screen on	0.69
Device downloading; phone connected with screen on	0.70

Table 4 lists the instantaneous current measurements recorded during various states of device operation. When the mobile phone was connected to the Raspberry Pi, it began charging and drew additional power from the battery. This unintended side-effect caused the battery to drain faster than expected. However, the battery provided several hours of operation after it was fully charged.

The Raspberry Pi 3 schematics are limited and do not include the USB controller and connections. Previous versions of the Raspberry Pi have direct connections between the power input 5 V line and the 5 V

line on the USB ports. However, the 5 V lines on the USB ports of the Raspberry Pi 3 are not powered when the device is powered without a bootable image available. This suggests that the USB controller may be able to disable the power output on the USB ports. Disabling the unnecessary power drain through the Raspberry Pi is not critical, but it would be useful for future applications of the hardware solution.

5. Conclusions

Mobile devices have complex attack surfaces and vulnerabilities that can be exposed and exploited when they connect to networks. Increasing device complexity and ubiquitous mobile access necessitate the development of new techniques for detecting and mitigating mobile device malware.

Sophisticated malware uses a variety of techniques to avoid detection and capture. Encryption and encoding have been used to evade signature-based detection for years. Self-destructing malware erases itself to avoid discovery during digital forensic investigations. Memory-resident malware that never uses non-volatile storage disappears when the device is shut down or rebooted.

These sophisticated malware features require novel detection and capture techniques. This chapter has described a new technique that enables the capture of memory-resident malware using live memory digital forensic tools (e.g., `LiME`). The automated capture technique enables the discovery and analysis of previously unknown exploitation techniques as well as the implementation of new mitigation strategies for vulnerable devices. Most importantly, the modifications required to implement the technique are minimal – the modified device contains the same vulnerabilities found in an unmodified version of the device.

A memory capture technique will not mitigate any vulnerabilities unless the captured malware can be analyzed successfully. Therefore, the capture technique is designed to support malware analysis. The captured images are compatible with the Volatility framework.

Future research will focus on developing improved guidelines and techniques for identifying malware in captured memory images. Additionally, memory images from normal devices and exploited devices will be compared in an attempt to automate malware analysis.

References

[1] H. Be'er, Metaphor: A (Real) Real-Life Stagefright Exploit, Revision 1.1, NorthBit, Herzliya, Israel (`raw.githubusercontent.com/NorthBit/Public/master/NorthBit-Metaphor.pdf`), 2016.

[2] R. Bejtlich, *The Tao of Network Security Monitoring: Beyond Intrusion Detection*, Addison-Wesley, Boston, Massachusetts, 2004.

[3] M. Brand, Stagefrightened? Project Zero, Google, Mountain View, California (googleprojectzero.blogspot.com/2015/09/stagefrightened.html), September 16, 2015.

[4] J. Drake, Stagefright: Scary code in the heart of Android, presented at the *Black Hat USA Conference*, 2015.

[5] G Data Software, G Data Mobile Malware Report, Threat Report: Q2/2015, Bochum, Germany, 2015.

[6] Z. Grimmett, J. Staggs and S. Shenoi, Categorizing mobile device malware based on system side-effects, in *Advances in Digital Forensics XIII*, G. Peterson and S. Shenoi (Eds.), Springer, Cham, Switzerland, pp. 203–219, 2017.

[7] C. Pfleeger and S. Lawrence-Pfleeger, *Security in Computing*, Prentice Hall, Upper Saddle River, New Jersey, 2007.

[8] Scientific Working Group on Digital Evidence, SWGDE Best Practices for Mobile Phone Forensics, Version 2.0, 2013.

[9] H. Sun, K. Sun, Y. Wang, J. Jing and S. Jajodia, TrustDump: Reliable memory acquisition from smartphones, *Proceedings of the Nineteenth European Symposium on Research in Computer Security*, part I, pp. 202–218, 2014.

[10] J. Sylve, A. Case, L. Marziale and G. Richard, Acquisition and analysis of volatile memory from Android devices, *Digital Investigation*, vol. 8(3-4), pp. 175–184, 2012.

[11] V. Thing, K. Ng and E. Chang, Live memory forensics of mobile phones, *Digital Investigation*, vol. 7(S), pp. S74–S82, 2010.

[12] Zimperium zLabs, The Latest on Stagefright: CVE-2015-1538 Exploit is Now Available for Testing Purposes, San Francisco, California (blog.zimperium.com/the-latest-on-stagefright-cve-2015-1538-exploit-is-now-available-for-testing-purposes), September 9, 2015.