HAL open science

Categorizing Mobile Device Malware Based on System Side-Effects

Zachary Grimmett, Jason Staggs, Sujeet Shenoi

To cite this version:

Zachary Grimmett, Jason Staggs, Sujeet Shenoi. Categorizing Mobile Device Malware Based on System Side-Effects. 13th IFIP International Conference on Digital Forensics (DigitalForensics), Jan 2017, Orlando, FL, United States. pp.203-219, 10.1007/978-3-319-67208-3_12 . hal-01716405

HAL Id: hal-01716405

https://inria.hal.science/hal-01716405v1

Submitted on 23 Feb 2018

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 12

CATEGORIZING MOBILE DEVICE MALWARE BASED ON SYSTEM SIDE-EFFECTS

Zachary Grimmett, Jason Staggs and Sujeet Shenoi

Abstract Malware targeting mobile devices is an ever increasing threat. The most insidious type of malware resides entirely in volatile memory and does not leave a trail of persistent artifacts. Such malware requires novel detection and capture methods in order to be reliably identified, analyzed and mitigated. This chapter proposes malware categorization and detection techniques based on measurable system side-effects observed in an exploited mobile device. Using the Stagefright family of exploits as a case study, common system side-effects produced as a result of attempted exploitation are identified. These system side-effects are leveraged to trigger volatile memory (i.e., RAM) collection by memory acquisition tools (e.g., `LiME`) to enable analysis of the malware.

Keywords: Mobile malware, memory-resident, categorization, system side-effects

1. Introduction

Critical vulnerabilities that affect large families of mobile devices make it imperative to develop new techniques for securing these devices against increasingly sophisticated attacks as well as for conducting forensic investigations. Investigating attacks on mobile devices requires the capture and analysis of evidence pertaining to attacks. However, the most insidious malware resides entirely in memory and does not create persistent artifacts. Memory-resident malware that removes itself from a mobile device after performing its malicious activities can evade capture and analysis by malware investigators, even after its presence has been detected by a user. Live memory acquisition is the only way to recover memory-resident malware from exploited devices.

This chapter introduces a taxonomy for categorizing mobile device malware based on observable system side-effects in an effort to stimulate the development of new methods for memory-resident malware detection, capture and analysis. The Stagefright family of vulnerabilities and exploits is used as a case study to assess the system side-effects that occur as a result of exploitation attempts of the `libstagefright` Android library. The chapter also describes how system side-effects produced by this malware can be used to trigger volatile memory captures for subsequent malware analysis efforts.

2. Live Memory Analysis of Mobile Devices

Live memory analysis refers to the capture and analysis of data stored in the volatile memory of a computer system or device. Numerous situations exist where important information only resides in volatile memory. These circumstances demand the application of techniques that can reliably and safely extract the information for forensic analysis.

Note that mobile devices have many more variations than computer workstations in their design and architecture. Beyond the obvious differences in design requirements due to their size and power constraints, mobile devices perform other unique tasks that require special consideration. Radio communications are highly sensitive to timing and poorly suited to sharing processing time with user applications. Moreover, user applications are equally ill-suited to execute on a real-time operating system that could enable reliable radio communications. As a result, most mobile devices contain two processors – an applications processor that handles user applications and a baseband processor that independently handles cellular communications (e.g., GSM, UMTS and LTE).

2.1 Information in Volatile Memory

The physical memory of a device essentially contains a snapshot of the recently-used information on the device. Recent web searches, text messages and session keys can all be recovered with access to physical memory [19]. This is why many devices now deny access to physical memory and restrict applications to their own allocated memory. Legitimate uses for physical memory dumps, such as debugging application crashes, are arbitrated by the operating system.

While a mobile device contains many types of evidence of interest in digital forensic investigations, some artifacts or data are unrecoverable if they are lost from volatile memory. These artifacts include cryptographic keys for unlocking encrypted containers and private messages sent via secure messaging applications. Digital forensic investigators

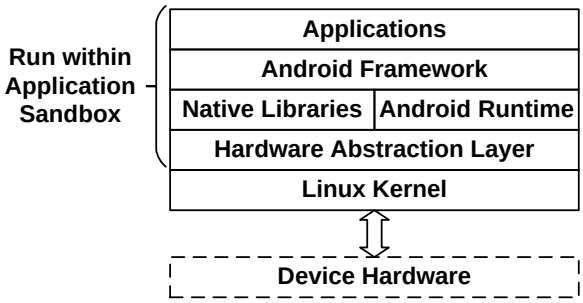

Figure 1. Android security architecture [1].

face a challenging trade-off – leaving a device powered on increases the risk of deleted data being overwritten while turning the device off risks losing evidence stored in volatile memory [13].

Another important use case for live memory analysis is malware detection and analysis. There are numerous examples of seemingly benign applications having malicious add-ons, including compromised applications that were pre-installed on some devices (e.g., Huawei G510 and Lenovo S860 smartphones) [9]. These applications do not require live memory analysis to detect and analyze. However, many of them act as Trojans that download and execute malicious code that may only exist in volatile memory. Understanding the threats posed by sophisticated mobile device malware requires deep analysis of the targeted hardware and operating system [7].

2.2 Memory Capture Techniques

Mobile devices have been developed with connectivity as the primary goal and have benefited from the lessons learned about the importance of securing connected devices. Mobile operating systems restrict user privileges to protect the devices and the network carriers.

Android uses long-standing Unix security concepts to provide application security – an application is limited to a "sandbox" and is assigned a unique UID that is used to apply and enforce user permissions. This ensures that only the Linux kernel has access to the process memory of more than one application.

Figure 1 presents the Android security architecture. The architecture limits software access to physical memory, a security improvement that prevents malicious applications from compromising other applications. For example, this prevents an infected social media application from having unfettered access to the memory of a banking application that may contain user account information and credentials. Under most cir-

cumstances, this is highly desirable behavior, but it also limits forensic access to physical memory. Memory-resident malware running in other applications or even system libraries may be effectively impossible to detect without system or hardware-level access to physical memory.

Several memory acquisition tools have been developed by digital forensic researchers. Thing et al. [19] have designed the `memgrab` tool, which parses process information in the filesystem (`/proc`) in order to locate process memory. Process tracing (via `ptrace`) is used to attach to a running process and suspend it while the process memory is being copied.

Sylve et al. [18] have developed the `Linux Memory Extractor` (`LiME`), a loadable kernel module that locates system memory and copies it to local storage or exfiltrates the memory over a TCP/IP network connection. `LiME` relies on parsing the kernel resource structure `iomem_resource` to identify physical memory locations in system RAM.

Stuttgen and Cohen [16] have attempted to create an even more general solution for creating forensically-sound images of live memory. Their solution leverages a minimal kernel module that can use other kernel modules to capture live memory. Another memory acquisition tool is TrustDump, which uses the ARM TrustZone to capture device memory in a manner that is completely transparent to the operating system [17].

3. Android Exploitation Techniques

Mobile devices have access to sensitive information (e.g., bank accounts, saved passwords and medical data), which has motivated the development and use of mobile device malware by criminals and hackers. Mobile operating systems prioritize reliability and availability so much that system processes restart as quickly as possible after a crash. The information saved when a process crashes is useful for debugging, but it is often insufficient to identify exploits. This section introduces exploitation techniques that impact how malware interacts with and resides in memory.

No single software solution can be expected to combat all potential malware on a mobile device. However, it is possible to design solutions that capture specific types of malware. Understanding how security mechanisms are defeated by malware is integral to a long-term effort to improve device security. In the short term, it enables researchers to discover and defend against current exploitation efforts. This short-term view of security is focused on finding and fixing existing vulnerabilities and benefits directly from efforts to capture previously-unidentified malware for analysis.

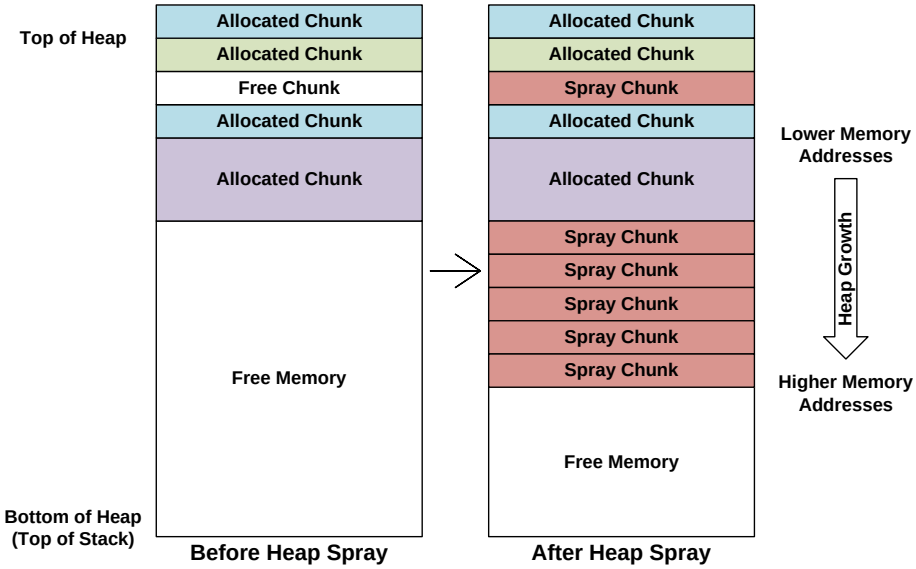

Figure 2. Heap spray example.

3.1 Heap Exploitation

Heap memory, or dynamic memory, enables a program to access and use memory as needed instead of requiring the program to request all the memory it will need at startup. Memory is allocated to a program in discrete chunks and is deallocated (freed) when it is no longer necessary. An attacker can manipulate the heap by performing specific allocations and deallocations that enable a vulnerability to be exploited. Heap exploitation leverages the control of heap memory to subvert a system. A program that does not properly verify or validate the use of dynamic memory is often vulnerable to multiple types of attacks.

Two common heap exploitation (or manipulation) techniques are: (i) heap spraying: and (ii) heap grooming:

- **Heap Spraying:** A heap spray involves a (generally large) number of allocations to place a designated chunk of memory into a specific location for later use (Figure 2). This leverages the tendency of a system to reuse and reorganize chunks in dynamic memory to avoid memory fragmentation. The specific location targeted by a heap spray is generally selected to be as reliable as possible while requiring no knowledge of the current dynamic memory layout.

  Programs routinely use dynamic memory to store user-controlled data – this only becomes a problem when the data is misused by

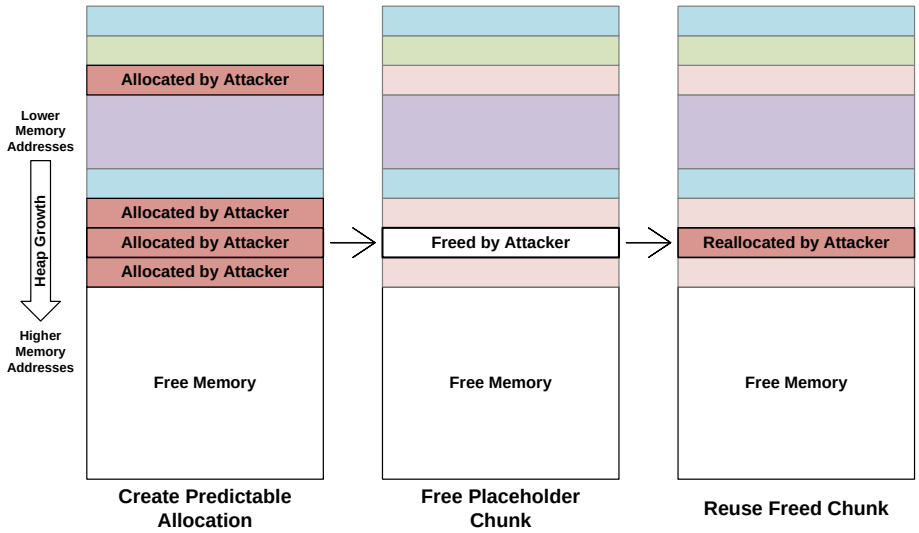

Figure 3. Heap groom example.

an exploit. A heap spray relies on an allocation of more memory than a system is expected to use. Such an allocation is noticeable because it involves an unusually large amount of memory. Several techniques have been developed to identify and prevent the anomalous use of dynamic memory [5, 11].

- Heap Grooming: Heap grooming uses allocations and deallocations to control an unspecified portion of the heap (Figure 3). When a program deallocates a chunk of memory and subsequently attempts to allocate another chunk of the same size, it is most efficient for the operating system to allocate the same piece of memory to the program. This behavior limits the impact of memory fragmentation without performing costly defragmentation techniques. Heap grooming takes advantage of the optimization by allocating a sequence of chunks and freeing a chunk in the middle of the sequence [15].

  Heap grooming uses far less memory than heap spraying and may display the behavior of a normal target program; this is because dynamic memory is intended to be allocated and deallocated as needed. Thus, heap grooming is more difficult to detect and prevent than heap spraying.

Heap manipulation techniques are not perfectly reliable. Systems with unexpected memory usage limit the probability of a heap spray or a heap

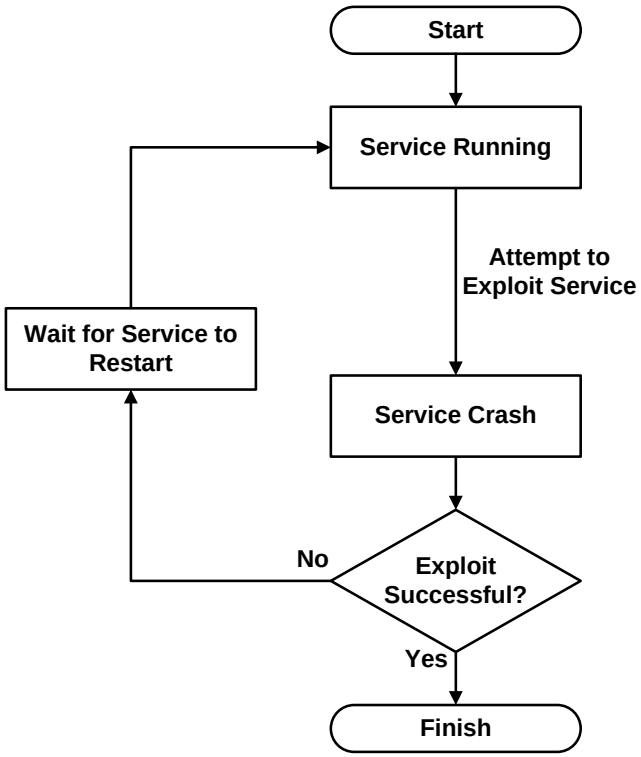

Figure 4. Brute-force execution.

groom succeeding. A heap exploit can be designed to maximize the chances of successful exploitation, but an alternative is to crash or reset a target process before attempting an exploit. In the case of a system that performs garbage collection after a process exits (or crashes), an attacker can assume that the process is in its initialized state and has predictable memory usage after it is restarted. Intentionally crashing a target process also serves another purpose – for attacks that require per-device or per-model adaptation (e.g., Stagefright), the presence of a vulnerability can be confirmed before any effort is made to develop an exploit for a particular model of device.

3.2 Defeating ASL Randomization

This section discusses techniques for defeating address space layout (ASL) randomization.

Brute-Force Execution. A brute-force execution attacks the same vulnerability repeatedly until the desired result occurs (Figure 4). The

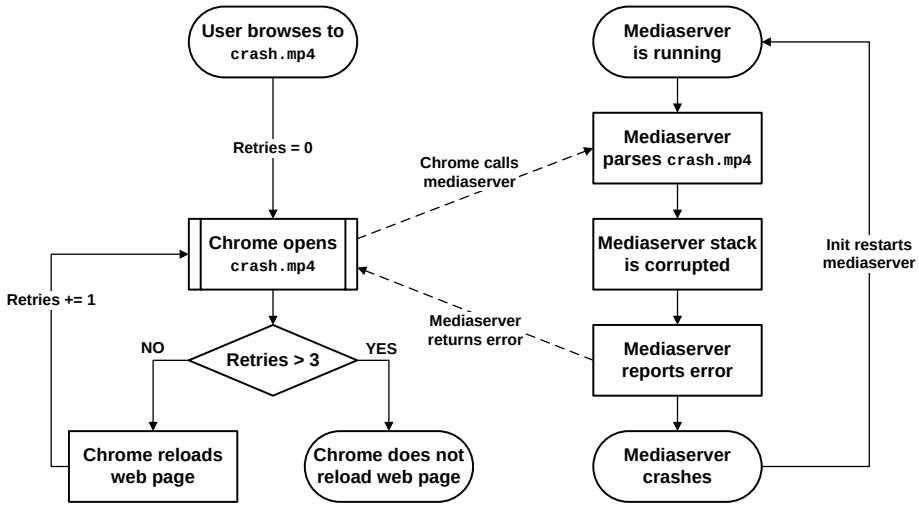

Figure 5. Google Chrome execution while parsing a crash vector.

initial Stagefright exploit (discussed below) repeatedly tries the exploit until address space layout randomization is defeated [6]. When the random offset address is not guessed correctly, the service crashes instead of executing the malicious code. Not all examples of brute-force execution either succeed or crash; it is possible for a program to merely return an error. The only requirement for brute-force execution is that the target returns to a vulnerable state after a failed exploit attempt.

The ability to mitigate brute-force exploitation attempts varies according to the system under attack. If reliability and availability are high priorities, rejecting or refusing to process information after a number of failed attempts may not be acceptable. It is worth noting that this mitigation is built into many applications to avoid getting stuck in an infinite loop. For example, as shown in Figure 5, Google Chrome stops the loading of a web page after four failed attempts. Unfortunately, a mitigation that is based on counting the number of failures could be subverted if an attacker succeeds before the maximum number of failures is exceeded.

Information Leaking. A memory leak involves the unintentional disclosure of information to an attacker. If the leaked information is sensitive, the leak itself may be the goal of an exploit. When memory address registers (e.g., stack pointer, heap pointer and program counter) are leaked, an attacker may gain useful information for exploiting the system [14]. Non-register addresses can enable an attacker to identify

the locations in memory where a system library has been loaded. Address space layout randomization can be subverted if an attacker leaks information and determines where the targeted libraries are loaded in memory.

Exploit developers often look for arbitrary read and write operations in vulnerable software to enable the development of reliable malware. These operations, also called primitives, are generic and useful; they are named after the read/write primitives that form the foundation of programming languages. The heap manipulation techniques discussed above can be used to leak information from memory in addition to enabling remote code execution. Limiting how a program handles sensitive information can mitigate memory leaks.

4. Stagefright Exploits

Stagefright is a family of exploits that target the `libstagefright` Android media-processing library [20]. Note that `libstagefright` refers to the media processing library and Stagefright refers to the family of vulnerabilities.

By exploiting integer overflow and memory corruption (heap overflow) vulnerabilities, a Stagefright exploit can be sent to an Android device and executed without the user's knowledge. The exploit triggers during preprocessing performed by `libstagefright` whenever a multimedia file is accessed, enabling it to be transmitted via text message, e-mail, web browsing or even when attempting to load a thumbnail of a malicious image saved on a device. The disclosure of the Stagefright exploits in conjunction with revelations that very few Android devices were receiving timely security patches resulted in new security update policies being released by Google [10] and Samsung [12]. This also prompted vulnerability researchers to focus on Android libraries, resulting in the discovery of additional vulnerabilities.

Multiple researchers have released proof-of-concept exploits for `libstagefright`. These exploits were generally released only after the exploited vulnerabilities were patched on applicable Android devices. The exploits frequently built on previous exploits by adding new capabilities or finding ways around the mitigation mechanisms. An examination of one of these exploits can reveal the nature of the vulnerability, but examining all of them can reveal how exploits evolve over time to combat mitigation efforts. A disclosed exploit can be used to demonstrate that the proposed modifications are successful at capturing malicious activity; however, a proposed solution should be resilient to changes in malware over time.

4.1 Zimperium zLabs

Drake [6] focused on the vulnerabilities in `libstagefright` because it is a privileged process (privileges inherited as a `mediaservice` process) that parses untrusted data. Additionally, `mediaserver` is started by the Android `init` process and is restarted whenever it crashes.

Drake chose to focus exclusively on MPEG4 file processing for fuzzing efforts; MPEG4 files are constructed in "chunks" that can be embedded inside each other. Parsing chunk code is complicated by the recursive MPEG4 file format and requires memory interactions that create vulnerabilities when unexpected sequences of chunks occur. An exploit developed for the CVE-2015-1538 vulnerability demonstrated that large Android frameworks incorporate assumptions that present significant risks to devices. The changes to the Android update policies discussed above occurred in response to this exploit, but before the details of the exploit were released to the public.

The initial Stagefright vulnerabilities presented by Drake [6] demonstrated that exploitation is possible through any vector that triggers media processing. This includes multimedia messages (MMS) that are automatically processed on receipt. Drake confirmed that the exploitation occurs before an alert is generated and displayed to a user. Effectively, an attacker could send a malicious multimedia message to a user and exploit the phone without any user interaction or notification.

The vulnerabilities were disclosed to Google before their public release, but most devices had not yet received the security updates for mitigating the exploits. Drake submitted patches to Google, but one patch introduced another vulnerability (CVE-2015-3864) that subsequent Stagefright efforts would exploit [8]. The proof-of-concept exploit [20] targeted an unspecified Nexus device (likely Nexus 5) running Android 4.0.4. It did not include an address space layout randomization defeat, but it achieved 100% reliability through repeated efforts because the `mediaserver` process is automatically restarted after it crashes due to a failed exploit.

4.2 Google Project Zero

Brand [4] leveraged the new vulnerability as the basis of a Stagefright exploit that targeted more recent versions of Android. Android versions 5.0 and later use a different memory allocation technique than older versions; the new allocation is based on `jemalloc` and necessitated changes to the heap grooming techniques used by the exploit [2]. Additionally, the address space layout randomization changes implemented in Android 5.0 made exploitation attempts less likely to result in remote

code execution. However, a proof-of-concept exploit revealed that these vulnerabilities were still present in newer Android devices.

Address space layout randomization successfully prevents an attacker from knowing exactly where shared libraries are loaded in memory, but this can be circumvented if the attacker can leak enough information to determine the memory layout. Alternatively, an attacker could guess where a library is loaded. The address space layout randomization implementation on Android devices only provides eight bits of entropy when the shared library (`libc.so`) is loaded; thus, an attack has a one in 256 chance of succeeding. Once again, because an unsuccessful attack crashes `mediaserver` and it automatically restarts, repeatedly trying the exploit eventually results in remote code execution. Brand [4] experimented with the exploit and discovered that successful exploitation took 30 seconds to a little over an hour.

4.3 NorthBit

Metaphor [3] is a Stagefright implementation that incorporates improved heap grooming capabilities and an address space layout randomization defeat. This exploit still targets the CVE-2015-3864 vulnerability added by Drake's patch, but it requires JavaScript execution to leak information and bypass address space layout randomization. This reduces the set of vectors vulnerable to the attack, but the exploit is more reliable and less dependent on predetermined library locations. This makes the exploit easier to adapt to other devices and it does not rely on any additional Stagefright vulnerabilities.

MPEG4 media files can include metadata (e.g., title, duration, copyright and lyrics) that is accessible by JavaScript. The same heap overflow vulnerability used to overwrite a function pointer for code execution can be leveraged to overwrite pointers in memory and enable access to arbitrary locations in memory. This primitive read operation overwrites the pointer to the duration value (an 8-byte integer) before returning metadata to the browser. However, because the browser requires the duration to be a signed 64-bit integer, negative or degenerate values are set to zero before they are reported to the browser. This limits the readable value to 32-35 bits of useful information after it is converted from microseconds to milliseconds.

The Metaphor exploit relies on the same address space layout randomization limitations as previous exploits – shared library modules are limited to a maximum address range of 256 memory pages. By iterating over these pages and performing a memory leak, an attacker could, in theory, identify the exact location of the targeted library. However, the

limitation on returned values prevents the reading of information that is normally used to identify a library (e.g., ELF header). Metaphor works around this limitation using `p_memsz` and `p_flags` as identifiers. These fields are relatively unique and are at known locations, so a lookup table can be created to match the read value to an expected value for the target module `libc.so`.

A proof-of-concept implementation includes server code that performs a memory leak until it determines the base address for `libc.so`; following this, it crafts and delivers the malicious media file. The media file performs the necessary heap grooming and overwrites a function pointer with an address controlled via heap overflow. Adapting the exploit to run on a new target is straightforward if an attacker has access to the version of `libc.so` running on the target device. This library can be extracted from a downloaded factory image or any device running the same version of the Android operating system.

These exploits demonstrate how quickly a discovered vulnerability can transition from a low-threat proof-of-concept to a sophisticated attack. Vulnerability researchers are paying much closer attention to Android frameworks, but the fear remains that a similar vulnerability could go unnoticed and result in large-scale compromise. The trade-off between a wide attack surface (initial Stagefright exploit) and a more sophisticated attack vector (Metaphor) is important from an attacker's perspective. It also plays a role in how mitigation mechanisms are developed and applied to resolve security problems.

5. Categorizing Malware by Behavior

This section presents a novel approach for capturing malware on an Android device for future analysis. A simple taxonomy is introduced that classifies malware based on the crash behavior of the exploited services.

Exploits that leverage brute-force techniques are designed with the expectation that a targeted service will crash multiple times. An attacker can intentionally cause a target service to crash in order to reset the memory of the service and create more predictable memory usage. Memory corruption exploits rely on sophisticated techniques (e.g., information leakage and heap grooming), but they may not be very reliable.

If an attacker designs an exploit to be as stealthy and as reliable as possible, it may not create side-effects that are detectable by the underlying system. A highly-reliable and well-hidden exploit could still be detected and captured on the rare occasion that it causes a crash. The best method for capturing the most sophisticated exploits is persistent

and continuous monitoring of volatile memory. However, no simple solution exists for finding an unknown malware sample in the large amount of data collected during a continuous data capture.

5.1 Malware Categories

Malware can be classified according to its intended and designed behavior. This classification enables the development of capture techniques that leverage the characteristics of each malware category.

- **User-Detectable Malware:** Not all malware is designed to avoid user detection – malware designed to intimidate or extort users intentionally disrupts and inconveniences victims. Mobile devices are now being targeted by "ransomware" that encrypts important files or locks users out of their devices until ransoms are paid. This category of malware is straightforward to detect and identify, but its disruptive behavior can make memory capture for malware analysis difficult.

- **System-Detectable Malware:** Malware can exhibit side-effects that are not obvious to a user, but can be detected by the underlying operating system. It is important to note that the focus is on side-effects that are explicit and well-defined. The side-effects include unreported service crashes, inappropriate application behavior and unexpected network connections. This category is not mutually exclusive with user-detectable malware; in most cases, the effects visible to a user are also apparent at the system level.

- **Inconspicuous Malware:** Inconspicuous malware does not create easily identifiable side-effects. This category includes malware that may be detectable through advanced analysis techniques (e.g., behavioral analysis and anomaly detection). Capturing this class of malware typically involves the collection of large amounts of data and eliminating the false positives.

5.2 Benefits of Malware Categorization

Categorizing malware according to observable side-effects facilitates the development of specialized detection techniques. These techniques are similar to heuristic analysis, but they rely on the results of attempted exploitation instead of analysis of the malware itself.

The Stagefright exploits demonstrate that mobile devices may hide side-effects (e.g., crash notifications and excessive memory paging) that are more noticeable on traditional computer workstations. The proposed categorization enables the capture and study of malware that relies on

the differences remaining undetected. More importantly, the categorization can also enable the detection of unknown malware that relies on similar assumptions.

Some exploitation mechanisms are tailored specifically to a target device – the Stagefright exploits leverage the same malicious media files to trigger vulnerabilities across multiple devices, but they require model-specific techniques to achieve code execution. Detecting device-specific exploitation mechanisms requires the development and deployment of solutions at the device model level. However, exploitation mechanisms (and their side-effects) that can be detected at the operating system level can be applied across multiple models of devices that run the same operating system.

5.3 Detecting Malware Side-Effects

As mentioned above, kernel-level access is necessary for a tool to arbitrarily dump memory that belongs to the operating system or other processes. `LiME` [18] is a loadable kernel module that can dump an image of the entire physical memory of a device with minimal impact. This makes `LiME` a useful tool for capturing malicious activity that cannot be precisely located in memory. `LiME` is well-suited to capturing large amounts of memory at one time, but not for consistent or continuous memory monitoring. This makes it useful in situations where suspicious activity can be detected (e.g., a service freezes or crashes unexpectedly), but its effectiveness against undetected attacks is limited.

System libraries (e.g., `libstagefright`) can be modified so that certain types of media are collected and saved before media parsing is performed. The number of captured files that are stored and the length of time they are maintained can be modified to suit the needs of researchers. If a device is monitored consistently, the files may be stored until they are analyzed. Conversely, if a device is only investigated in the event of a suspected compromise, then the stored files have to be managed because limited space is available on the device. However, changes made to system libraries increase the risk that an exploit that targets the libraries will no longer behave as expected. Consequently, this research has focused on modifications that do not alter common services.

Some libraries on Android devices are designed to support system and application developers. The debugging daemon (`debuggerd`) creates "tombstones" when an application or library crashes; these tombstones contain useful system information and program backtraces from the time of the crash. Figure 6 demonstrates how `debuggerd` is associated with every dynamically-linked executable. The executables specify the linker

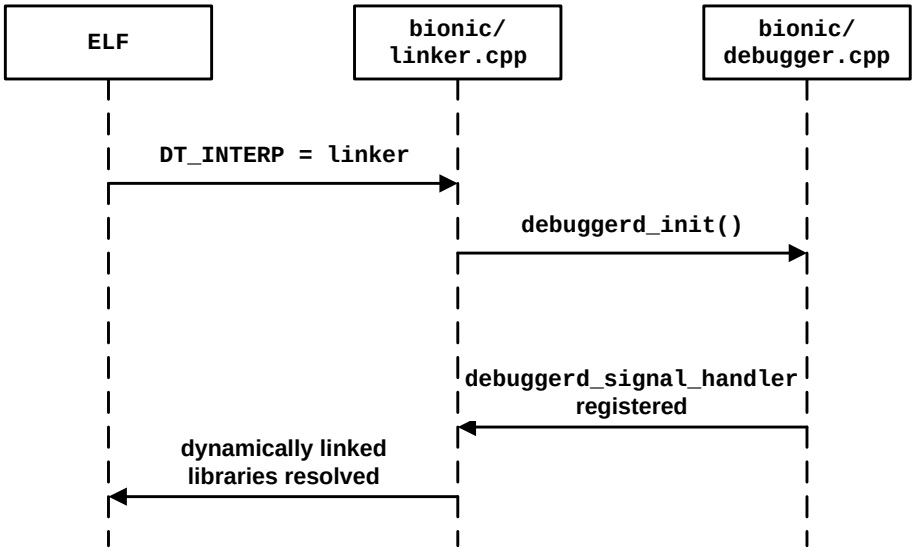

Figure 6. `debuggerd` handler setup.

`DT_INTERP` used to interpret the included symbols. Following this, the linker ties handlers for each signal to `debuggerd` before returning with the library symbols resolved.

The signal handlers are responsible for generating crash information. System properties can be configured to enable or disable additional debugging capabilities. The property `debug.db.uid` causes `debuggerd` to suspend a crashing process and attach `gdb` to the process – this enables a user to connect using `gdb` and actively debug the process before it crashes. (Note that the `uid` property is replaced with `wait_for_gdb` in newer Android devices.) The process remains suspended until the user depresses the "volume down" button or uses `gdb` to resume the process.

The `debuggerd` daemon is a useful tool for debugging Android platform code, but its backtrace and memory dump functionalities are poorly suited to analyzing exploits. Corrupted addresses in stack memory and unexpected register values can cause `debuggerd` to miss portions of memory that are relevant to malware analysis. To overcome this limitation, `debuggerd` may be modified to support additional functionality when a crash occurs. The limited memory capture functionality of `debuggerd` can also be enhanced with support for automatically launching the `LiME` capture process when crashes occur. The stated requirement that modifications to common libraries should be avoided can be overlooked for `debuggerd` because it is explicitly prevented from attempting to debug itself by design.

6. Conclusions

Combating sophisticated malware requires novel detection, capture and mitigation techniques. This research has proposed malware detection techniques based on measurable side-effects in an exploited device. Categorizing malware to identify common side-effects enables the automated capture of memory-resident malware using digital forensic tools for live memory acquisition. The automated capture technique enables digital forensic investigators to discover and analyze previously-unknown exploitation techniques and to implement new mitigation strategies for vulnerable devices. Most importantly, the proposed modifications that make the technique possible are minimal and device-independent.

References

[1] Android Open Source Project, Security (`source.android.com/security`), May 22, 2017.

[2] P. Argyroudis and C. Karamitas, Exploiting the `jemalloc` memory allocator: Owning Firefox's heap, presented at the *Black Hat USA Conference*, 2012.

[3] 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.

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

[5] M. Cova, C. Kruegel and G. Vigna, Detection and analysis of drive-by-download attacks and malicious JavaScript code, *Proceedings of the Nineteenth International Conference on World Wide Web*, pp. 281–290, 2010.

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

[7] J. Edmonds, Cell Phone Reverse Engineering and Malware Analysis, Ph.D. Dissertation, Tandy School of Computer Science, University of Tulsa, Tulsa, Oklahoma, 2012.

[8] Exodus Intelligence, Stagefright: Mission Accomplished? Austin, Texas (`blog.exodusintel.com/2015/08/13/stagefright-mission-accomplished`), August 13, 2015.

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

[10] A. Ludwig and V. Rapaka, An Update to Nexus Devices, Google, Mountain View, California (`officialandroid.blogspot.com/2015/08/an-update-to-nexus-devices.html`), August 5, 2015.

[11] P. Ratanaworabhan, B. Livshits and B. Zorn, NOZZLE: A defense against heap-spraying code injection attacks, *Proceedings of the Eighteenth USENIX Security Symposium*, pp. 169–186, 2009.

[12] Samsung Electronics, Samsung Announces an Android Security Update Process to Ensure Timely Protection from Security Vulnerabilities, Press Release, Suwon, South Korea, August 5, 2015.

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

[14] F. Serna, The info leak era of software exploitation, presented at the *Black Hat USA Conference*, 2012.

[15] A. Sotirov, Heap feng shui in JavaScript, presented at the *Black Hat Europe Conference*, 2007.

[16] J. Stuttgen and M. Cohen, Robust Linux memory acquisition with minimal target impact, *Digital Investigation*, vol. 11(S1), pp. S112–S119, 2014.

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

[18] 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.

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

[20] 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.