Memory forensics: The path forward

Article *in* Digital Investigation · January 2017

DOI: 10.1016/j.diin.2016.12.004

CITATIONS
30

READS
3,049

2 authors:

Andrew Case

17 PUBLICATIONS **333** CITATIONS

SEE PROFILE

Golden G. Richard III

Louisiana State University

86 PUBLICATIONS **1,533** CITATIONS

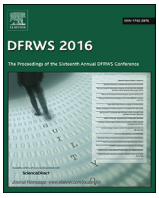
SEE PROFILE

Digital Investigation xxx (2016) 1–11

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin



Research Summary

Memory forensics: The path forward

Andrew Case [a], Golden G. Richard III [b, *]

[a] New Orleans, LA, United States
[b] Center for Computation and Technology and Division of Computer Science and Engineering, Louisiana State University, Baton Rouge, LA, United States

ARTICLE INFO

Article history:
Available online xxx

Keywords:
Memory forensics
Computer forensics
Memory analysis
Incident response
Malware

ABSTRACT

Traditionally, digital forensics focused on artifacts located on the storage devices of computer systems, mobile phones, digital cameras, and other electronic devices. In the past decade, however, researchers have created a number of powerful memory forensics tools that expand the scope of digital forensics to include the examination of volatile memory as well. While memory forensic techniques have evolved from simple string searches to deep, structured analysis of application and kernel data structures for a number of platforms and operating systems, much research remains to be done. This paper surveys the state-of-the-art in memory forensics, provide critical analysis of current-generation techniques, describe important changes in operating systems design that impact memory forensics, and sketches important areas for further research.

Introduction

Traditional storage forensics comprises a set of techniques to recover, preserve, and examines digital evidence and has applications in a number of important areas, including investigation of child exploitation, identity theft, counter-terrorism, intellectual property disputes, and more. Storage forensics tools are focused primarily on "dead" analysis, typically using bit-perfect copies of storage media. From these copies, deleted files or file fragments are recovered, patterns of file access are determined, past web browsing activity is observed, etc. Over the past decade, a number of factors have contributed to an increasing interest in *memory forensics* techniques, which allow analysis of a system's volatile memory for forensic artifacts. These factors include a huge increase in the size of forensic targets, larger case back-logs as more criminal activity involves the use of computer systems, the use of forensics techniques in incident response to combat malware, and trends in malware development, where malware now routinely leaves no traces on non-volatile storage devices. Importantly, memory forensics techniques can reveal a substantial amount of volatile evidence that would be completely lost if traditional "pull the plug" forensic procedures were followed. This evidence includes lists of running processes, network connections, fragments of volatile data such as chat messages, and keying material for drive encryption.

While there has been tremendous progress in building advanced memory forensics tools since the first rudimentary techniques were developed around 2004, much work remains to be done in this exciting research area. This paper surveys the state-of-the-art in a number of areas in memory forensics, including acquisition and analysis, and attempts to clearly lay out the research challenges that lie ahead. These challenges include not only work that remains to be done, such as better analysis techniques for user-level malware, but also fundamental shifts in how memory forensics tools are designed and how they operate, to accommodate significant changes in operating systems design.

Area of focus – memory acquisition

Historical approaches to memory acquisition

The ability to acquire volatile memory in a stable manner is the first prerequisite of memory analysis. Traditionally, memory acquisition was a straightforward process as operating systems provided built-in facilities for this purpose. These facilities, such as */dev/mem* on Linux and Mac OS X and *||.|Device||PhysicalMemory* on Windows, provided administrator-level users direct access to physical memory.

On modern systems, such facilities are generally not available, however, due to security concerns. In particular, both */dev/mem* and *PhysicalMemory* allowed for read and write access to physical memory (CrazyLord, 2002). This allowed malware not only to steal the contents of kernel memory, but also to modify it. The notorious

* Corresponding author.
E-mail addresses: andrew@dfir.org (A. Case), golden@cct.lsu.edu (G.G. Richard).

http://dx.doi.org/10.1016/j.diin.2016.12.004
1742-2876/

Phalanx2 rootkit (Case, 2012) leveraged */dev/mem* in this manner as do many other malware variants across operating system versions.

Beyond the security issues, these built-in facilities were becoming of limited use to investigators as systems under investigations were rapidly moving towards:

- Having multiple CPU cores and/or physical processors installed
- Increasing amounts of RAM
- Operating system adoption to the demands of scale

Even if the kernel devices such as */dev/mem* were available, the implementations are not safe for memory acquisition on multicore/multi-CPU systems, as races to map, remap, and unmap pages can result in kernel instability. Furthermore, since the acquisition tools reading from these interfaces are executing in userland, in-kernel synchronization primitives can't be used to solve the problem. Thus kernel modules must typically be loaded to allow memory to be acquired and on multicore/multi-CPU systems, the kernel modules must be carefully designed to pay special attention to the operating system-specific rules governing how and when kernel mode code can be interrupted (generally known as kernel preemption) (Stüttgen and Cohen, 2014).

The continuously increasing amount of RAM installed in systems leads to page smearing (Carvey, 2005), which is an inconsistency between what the state of memory as described by the page tables is versus what is actually in those pages of memory. This issue occurs due to the time lapse between when the page tables are acquired versus when data is acquired in other portions of RAM. Smearing will be discussed in full detail in the following sections.

Current issues — page smearing

The following sections describe current approaches to acquisition across all major operating systems, along with the limitations of these approaches. Each section is structured to describe the state of the art, its limitations, and future directions to improve acquisition techniques and procedures. We start with page smearing as it is one of the most pressing issues.

Current state

As mentioned above, page smearing is an inconsistency that occurs in memory captures when the acquired page tables reference physical pages whose contents changed during the acquisition process. In our experience, this problem is commonly encountered on systems that have 8 gigabytes or more of RAM installed as well as systems that are under heavy load. Unfortunately, systems with less than 8 gigabytes of RAM are increasingly uncommon and servers frequently have from 16 gigabytes of RAM to hundreds of gigabytes. The increasing amount of RAM installed in computer systems means that nearly *all* captures will contain at least some amount of smear. Depending on where the smear occurs, this can result in undesirable results of varying degrees of severity, from memory pages belonging to one process being assigned to another in the view of the memory analysis tool, to corrupted kernel data structures. Unfortunately, memory analysis tools and frameworks have no method to automatically detect smearing as there is only one source of data for address translation - the smeared page tables themselves.

The move to the cloud and the adoption of local cloud computing models has led to an increase in system utilization across servers. As recommended by both the Amazon AWS (Amazon ec2 container serv, 2016) and Google Cloud (Scaling based on cpu or l, 2016) documentation, running systems at 75% load achieves the greatest balance of CPU utilization without over-utilization as well as cost savings. An industry-wide shift to every server acquired running at 75% capacity with 16GB+ of RAM is a very different model than the one used to conceive current generation memory acquisition algorithms.

Issues and limitations

With the exception of attempting to acquire memory as quickly as possible, acquisition tools currently do not make any effort to detect or work around smearing. Analysts routinely try to work around smearing by leveraging hypervisor capabilities when access to the hypervisor host is possible. As discussed in Ligh et al. (2014), virtualization technologies such as VMware, HyperV, and VirtualBox provide the ability to acquire guest memory VM from the host. This acquisition can be performed “instantly” by leveraging hypervisor-specific features, such as snapshots and suspended states, to freeze the guest in-place. This prevents smearing from occurring as the guest can no longer make modifications to memory, and the analyst can simply copy the saved memory state from the hypervisor's file system. This approach is not universally usable, however, as obviously not all systems are virtualized. Furthermore, even in virtualized environments the analyst does not always have access to the physical host, such as in Amazon's AWS or Microsoft's Azure. There may also be issues with temporality suspending a running virtual machine as it may affect production performance and stability.

Future directions

The issues created by page smearing necessitate that memory acquisition tools take smearing into account during the acquisition phase. To this end, we propose several methods that may meet this need.

Leveraging virtual machine hardware extensions. Our first proposed method is a modern implementation of the BodySnatcher tool (Schatz, 2007). BodySnatcher attempted to ‘freeze’ the running operating system and load a small, second operating system at runtime. This was performed without the support of hardware-based virtual machine support as that technology was not yet in use.

We envision that a modern approach to BodySnatcher could instead leverage hardware based virtualization to cleanly:

- Insert a new operating system “under” the existing one
- Freeze the existing operating system
- Write memory of the frozen operating system to removable media or the network

Blue Pill (Rutkowska and Tereshkin, 2008) is a famous example of leveraging this technology to implement rootkit functionality, and the same approach could be taken for defensive acquisition purposes. Besides acquisition from native hardware, this approach would also work for virtual machine guests where the analyst did not have host access, as hardware virtualization extensions allow for nesting of virtual machines. The downside is similar to those of leveraging traditional hypervisors in that network connections and other activity would be frozen for the duration of the acquisition. This is usually not an issue for end-user systems, but is often unacceptable for production servers.

Smear-aware acquisition tools. Our second proposed method is for acquisition tools to become aware of changes to the page tables as acquisition is performed. Unfortunately, there is no single place in the kernel that could be monitored for such changes, as applications and kernel drivers continuously allocate and deallocate memory as well as making changes to memory as the acquisition tool runs.

Please cite this article in press as: Case, A., Richard, III, G.G., Memory forensics: The path forward, Digital Investigation (2016), http://dx.doi.org/10.1016/j.diin.2016.12.004

To work around this issue, acquisition tools could start analysis by parsing the page table entries of every context, which includes the kernel's context as well as the context of every process. This gathered data would then be saved to the same medium as where the memory capture will be written. Next, memory would be acquired as it currently is. Upon completion, the page tables would then be enumerated and written out to a separate file on the storage medium.

Having the state of the page tables both before, during, and after acquisition would allow analysis frameworks to decide what to do with pages that have changed. For example, the framework could simply choose to discard all pages that were modified when performing analysis of active data (e.g., running Yara rules across process memory). The framework could also decide to incorporate the changed page, but print debug messages to users that data has changed at particular addresses. Advanced approaches could include incorporating historical copies of data from places in memory that used to hold the data a process referenced. We believe there are many avenues of research in this area that could be beneficial to sound memory acquisition.

Current issue – incorporation of non-resident pages

Current state

Besides page smearing, page swapping and demand paging are currently the biggest obstacles to complete acquisition of memory. Page swapping is mechanism that allows for an operating system to use more memory than is actually available in RAM. To achieve this, the operating system stores excess pages in a secondary source, normally in a local file system, until those pages are needed again. Demand paging is a mechanism whereby the operating system does not bring information from files on disk into memory until they are absolutely needed, usually as a result of a read or a write operation to a portion of a file.

Combined, these issues prevent analysts from gaining a complete picture of what was happening on the system at the time of acquisition. As discussed in Ligh et al. (2014), the pages that are commonly swapped to disk are those that are associated with userland processes and not the kernel. Such pages will have revealing details of user activity, such as web browsing, command line activity, DNS requests and responses, and email activity that is often crucial to investigations. Demand paging causes its own issues as many pages of the running executable and its shared libraries (DLLs) will be on disk and not in memory. This can prevent complete binary analysis of components that may have malicious functionality or that have been a victim of code injection.

Issues and limitations

The seemingly obvious solution to the issues of page swapping and demand paging is for the memory acquisition tool to first acquire memory and then acquire the page file(s) along with any actively referenced files in the file system. Unfortunately, applying this approach to real world investigations is not as easy as it seems.

To begin with, as mentioned previously, acquisition tools do not currently handle page smearing in main memory. This problem is only multiplied then when applied to acquisition of the page file. Just as page table entries and contents in main memory can be smeared, so can entries in the paging file. In fact, the entries are usually further smeared as the page file is acquired after memory acquisition is finished in order to follow the Order of Volatility (Farmer and Venema, 2005). This acquisition flow nearly guarantees that pages in the swap file will have changed from when the page tables that describe the swap file contents were acquired. The increase in RAM of common systems exacerbates this problem as Windows creates paging files whose size is related to the size of RAM. This can make acquisition of the page file take a substantial amount of time, especially when acquisition is performed over the network.

Accounting for demand paging also presents several problems. First, in order to determine which files are being accessed by running code requires the acquisition tool to essentially perform memory forensics. In particular, it would need to parse the handle (file descriptor) tables as well as file-backed memory mappings of each process in order to gather a list of files to acquire. This is a substantial amount of logic to add to the acquisition tool, especially if support for a wide variety of operating systems and operating systems versions is important.

Second, in order to maintain the Order of Volatility, all memory should be acquired first, and then the handle tables and memory mappings parsed second. This will necessarily lead to incomplete and inconsistent data as processes will have opened and closed handles during this time as well as mapped and unmapped files. There is also the chance of processes terminating or being created during acquisition. All of these cases would lead to missing files.

The last issue is that even if the acquisition tool could gather a complete list of files needed, a blacklist of files not to acquire would be needed. As an example on Windows, the kernel thread process (System) can have handles open to registry hives, the hibernation file, swap files, and others. These files can be quite large and are often not the targets of demand paging. Furthermore, processes such as database servers will have handles opened to huge files that are often not relevant and whose collection would substantially slow down acquisition speed. Fine-tuning of which files to collect and which not to collect would require intensive research and any missing entries from the blacklist could jeopardize acquisitions.

Future directions

Due to the importance of data in the swap file, we feel that it is crucial for the acquisition process to correctly acquire the swap file as well as for acquisition to then incorporate analysis of it. In this subsection we describe our proposed methods for acquisition tools to acquire the swap file with limited (or no) smear, and in Section "Area of focus – memory analysis" we describe how analysis frameworks can seamlessly integrate swap.

Unlike page smearing directly in RAM, smear related to page file acquisition occurs only through the operation handlers for reading and writing to the swap file. If an acquisition tool were to monitor these operations, it could then decide to passively or actively account for pages coming in and out of the swap file as acquisition occurs.

In passive mode, the acquisition driver could monitor reads and writes to the page file and record the physical offsets on which they occurred. This monitoring would start from when the driver is loaded until both volatile memory and the page file(s) from disk were acquired. The set of pages affected by swapping could then be stored in metadata associated with the capture. Memory analysis tools that integrated swap file analysis could then leverage this metadata to ignore regions that changed. This would effectively stop smeared pages from being incorporated into analysis results.

In active mode, the acquisition driver could go a step further and instead of simply recording which pages are being smeared, it could also read the data currently in the swapped position before allowing it to be overwritten. The memory acquisition tool could then write the recovered data into a separate store along with metadata describing its physical offset. Memory analysis tools would then read the list of pages that were affected by smear and substitute in the recovered data on reads to the affected offsets. With flexible memory analysis frameworks, such as Volatility, this would be a minor update to how address translation already occurs. This process of saving pages before they are overwritten and

Please cite this article in press as: Case, A., Richard, III, G.G., Memory forensics: The path forward, Digital Investigation (2016), http://dx.doi.org/10.1016/j.diin.2016.12.004

incorporating them into analysis would remove the effects of pagefile-based smearing.

Current issue – changes to Windows hibernation file analysis

Current state

System hibernation is the process of a computer suspending itself in a manner that allows the user's session to be resumed quickly but without requiring power between uses. During the hibernation process, the operating system writes a complete capture of RAM to the local file system in order to allow for a full power down. This is contrast to system 'sleep' modes, which put the system into a low power mode that maintains the state of system RAM.

Since Windows XP, memory forensic analysts have leveraged the contents of RAM written during to disk during hibernation, referred to as 'hibernation files', in order to acquire a historical capture of memory. Analysis of these files was initially made possible due to research performed by Matthe Suiche and presented at Black Hat 2008 (Suiche, 2008).

Issues and limitations

Beginning with Windows 8, Microsoft substantially changed both the on-disk format of hibernation files as well as the operations performed upon it during system resume. Before Windows 8, after the system resumed, the only data modified in the hibernation file was the header. This allowed memory analysis tools to fully analyze the RAM contents contained within the file. Starting with Windows 8, however, this procedure is modified and the header is left in-tact, but the rest of file's contents is overwritten with zeros (Sylve et al., 2016). This makes analysis of hibernation files from running systems impossible on modern Windows versions.

Future directions

Because the hibernation file is zeroed when the system boots, there is little that can be done during acquisition or analysis to recover memory artifacts from hibernation files acquired from live Windows systems running Windows 8 or later. Instead, we propose that live acquisition scripts and procedures be altered to exclude acquisition of these files. This will substantially speed up the acquisition process as Windows hibernation files are of roughly the same size as the amount of RAM installed. Avoiding analysis of these files will also prevent wasted analyst time and effort. Furthermore, we suggest that developers of memory analysis tools and frameworks provide end-users with explicit warning messages when users try to analyze hibernation files from such systems. This will again prevent analysts wasting time trying to troubleshoot their tools and contacting support for clarification.

Current issue – Windows 10

Current state

In addition to the changes in hibernation analysis, Windows 10 also introduced several other features that affect memory forensics. This section focuses on issues that currently are under-researched or for which no public research exists. Due to the rapid adoption of Windows 10 throughout corporate enterprises, it is vital that the digital forensics community better understand this new operating system and proper memory forensic approaches applicable to it.

Issues and limitations

There are currently at least two areas where further research for Windows 10 acquisition is needed.

Device Guard. The first, Device Guard (Device guard deployment guide, 2016), is a major architectural change to Windows whose purpose is to protect critical components of the operating system from tampering by malware. It accomplishes this protection by transparently running the user-facing operating system inside of a virtual machine guest. The hypervisor host, which the user has no access to, then protects memory of critical processes, such as LSASS, as well as monitors for malicious changes to the system, such as malware overwriting core kernel components. Even kernel-level malware will be defenseless against these protections as the security code is running in an isolated environment with higher privileges. This is in sharp contrast to the traditional arms race between kernel-level malware and kernel-level security agents that both ran within the same security boundary. The implications for memory acquisition are that without a break out of virtualization exploit, only the memory of the guest VM can be acquired and significant portions of the operating system's state are inaccessible to acquisition tools.

Swapfile.sys. The next major change involves the addition of a new virtual memory file, swapfile.sys (Geek, 2015). This file was created to handle swapping of Microsoft's new style of applications, now referred to as "Universal Windows Platform Apps", but also as "Windows Store Apps", "Metro Apps", and "Windows Modern Apps". With this new swapping strategy, instead of individual pages from a Metro App being swapped out into pagefile.sys, the entire working set of the application is written into a new file, swapfile.sys. In Windows terminology, an application's working set is all of the addressable pages that a process currently has assigned to its address space.

Universal and Metro Apps. New Metro Apps not only introduced changes related to swapping of virtual memory, but they also introduce a rich set of new APIs that allow for interacting with touch screens and other peripherals that are not present on traditional devices (MSDN, 2016a). These new APIs allow for legitimate applications to monitor a wide range of user behavior and hardware interactions in order to respond to such events.

Future directions

All of the discussed Windows 10 changes necessitate research in order for memory forensic analysts to have a full understanding of an investigated system's state as well as to understand how malware may abuse these new features.

Device Guard. While Device Guard's security enhancements will prevent traditional malware from accomplishing its goals, it also presents a major stumbling block for analysts. Since memory acquisition tools run inside the protected guest, analysts have no insight into what occurs at the hypervisor level or inside the guest(s) that the hypervisor creates in order to run the security monitors. It seems inevitable that sophisticated attackers will escape this guest virtual machine into the hypervisor, and at the current time, forensic investigators will have no means to track this activity.

Swapfile.sys. The addition of a new file that contains swapped virtual memory compounds the issues described previously in regards to smear and file acquisition. Furthermore, this new file will necessitate changes to how analysts approach analysis of swapped data. Currently, many analysts are trained to analyze pagefile.sys in order to find records of user and other system activity. With the addition of swapfile.sys, it appears as if all of the pages belonging to processes will instead be stored in a separate source. Considering that common applications, such as browsers, document viewers and editors, and chat applications are all being moved to the universal platform, it is reasonable to assume that

Please cite this article in press as: Case, A., Richard, III, G.G., Memory forensics: The path forward, Digital Investigation (2016), http://dx.doi.org/10.1016/j.diin.2016.12.004

substantial amounts of valuable data will no longer be in pagefile.sys.

Universal and Metro Apps. While the introduction of the new Metro APIs and interfaces are meant for legitimate purposes, malware has historically abused such APIs for keylogging, monitoring web cameras and microphones, and a wide range of other nefarious purposes (Ligh, 2012). We expect that malware authors will begin to adopt their toolkits to these new APIs as they are not currently monitored by endpoint agents and memory forensics toolkits. Monitoring of these interfaces will require research and development time by forensics tool developers, which gives attackers a gap in time in which to operate undetected.

Linux and Android acquisition

Current state

Linux and Android systems present unique memory forensic challenges. Unlike Windows and Mac OS X, where a memory acquisition kernel driver can be compiled for a broad range of kernel versions, Linux kernel modules need to be compiled for every version and subversion of the kernel. This close dependency on specific kernel versions combined with the speed of kernel development and the number of distributions, such as Ubuntu, Debian, RedHat, CentOS, and OpenSuSe, leads to a situation where complete Linux version support would encompass thousands of kernel modules. Creating and maintaining such a database requires dedicated support by a team of engineers as well as automation of checking for new updates and building of new operating system version kernel modules. There is also a considerable amount of monetary and infrastructure needs to host the database of profiles online in a world wide accessible manner.

There are currently no open source projects that dedicate this amount of effort to a database of Linux profiles. On the commercial side, Threat Protection for Linux (Garner, 2016), formerly known as Second Look, maintains a database of thousands of kernel modules and profiles. This database is only made available to customers and requires a mix of engineers and automated processing to keep up-to-date.

Issues and limitations

Leveraging a kernel module database. While useful in many instances, the dependency of a memory analysis tool or framework on a database of all existing kernel modules has several shortcomings. The first is that this database must be constantly updated. While it may be possible for a vendor to keep such a database up-to-date, it is often difficult for users of the associated tool to stay in sync with the updates. The most common issue is that many forensic systems are permanently disconnected from the Internet to avoid leakage of evidence. This makes updating systems a manual process, which usually involves burning CDs/DVDs and then transferring them, and few investigators will want to perform this process on a daily or weekly basis.

The second issue with this approach is that there is no fallback for systems running custom compiled kernels. While generally thought of as something only done by enthusiasts, these custom kernels are increasingly found in the field as vendors create kernels for highly specific use cases such as single purpose hardware devices as well as high performance systems.

The last issue is that it is not usable in incident response scenarios where the kernel versions of the systems to be acquired unknown. To leverage the database during these scenarios, a few steps must occur before acquisition can actually begin. First, the correct kernel module must be identified. This would ideally be done via an invocation of `uname -a` by the investigator, but might also require loading a separate tool to uniquely identify the version. Second, the identified kernel module must be obtained. In a situation where the entire database isn't mirrored to the investigator, such as in the Second Look approach, the individual module must then be downloaded. This shouldn't be done on the system under investigation, however, as it would be noisy and pollute data. Instead, a second system must be used to download the module and then it could be transferred to the system in question. Only after these steps could acquisition begin. This process has several significant disadvantages, including 1) it assumes the investigators are in an environment where Internet connectivity is allowed 2) this process does not scale, as it requires manual intervention and 3) the time between system version identification and acquisition starting leaves a time gap for attackers to take notice and act.

Leveraging existing kernel modules. A unique approach to avoiding the need for a new kernel acquisition model for every kernel version was presented in Stüttgen and Cohen (2014). In this paper, the issues with building a kernel module for every kernel were discussed, along with how the kernel verifies that a module to be loaded matches the running the kernel. The paper then discusses how, in order to bypass this check and still keep the system stable, a copy of an existing kernel module is infected with the acquisition algorithm. This allows memory acquisition code to be loaded into the kernel, since the existing module will have the correct metadata to pass the version check and also for acquisition to be in a kernel version independent manner.

This approach has disadvantages as well, however. As described in detail in the paper, in order for the acquisition process to remain stable, the injected acquisition algorithm is extremely simple. Simplicity is necessary to avoid leveraging APIs or using data structures whose implementation or layout changes between versions. The algorithm works by creating a character device, which makes the driver's functionality available to userland, implementing the read handler of the device driver so that the userland component can read from it, and implementing a page remapping algorithm in order to allow physical memory to be read by userland.

In contrast to this approach is the one used by LiME (Sylve et al., 2012), the most widely used acquisition tool for Linux. LiME provides a rich set of capabilities, such as the ability to write to local disk or over the network. LiME is also unique in that it performs acquisition directly from the kernel. As shown in Sylve et al. (2012), acquiring memory directly from the kernel, and as a result bypassing the need for thousands of context switches, leads to memory captures with a substantially larger percentage of the original data versus approaches that acquire through kernel-to-userland facilities.

LiME is not a perfect solution, however, as it requires its kernel module to be built for every version of the Linux kernel. Also, there are features missing that mature memory acquisition tools provide, such as file compression and over-the-network encryption, which would be difficult and error prone to implement in the kernel. This means that LiME would likely have to be redesigned to support userland facilities where such operations are straightforward. This would negatively affect its ability to acquire nearly pristine copies of memory.

Leveraging /proc/kcore. The loss of */dev/mem* led many in the digital forensics community to assume that all future acquisitions of Linux memory would require kernel drivers. This led to the development of tools based on kernel modules, such as fmem (niekt0 and fmem, 2011) and LiME. This assumption is not entirely valid, as */proc/kcore* still exposes the kernel's virtual memory space. First discussed in a digital forensics context by Burdach (2006) in 2006, on 64-bit

Please cite this article in press as: Case, A., Richard, III, G.G., Memory forensics: The path forward, Digital Investigation (2016), http://dx.doi.org/10.1016/j.diin.2016.12.004

systems */proc/kcore* allows full memory acquisition as the entire physical address space of the system is mapped into virtual memory. This is not possible on 32-bit systems due to virtual address space size constraints.

As with other solutions, */proc/kcore* does not solve the issues on its own. It is limited to only 64-bit systems and furthermore, its presence on a system is dependent on an optional kernel configuration option. We have encountered systems both with and without */proc/kcore*'s existence. The */proc/kcore* interface also suffers from the same issues with page mapping, remapping and unmapping as other interfaces like */dev/mem*, which can result in kernel instability during memory acquisition on multicore/multi-CPU systems.

Avoiding acquisition modules hinders structured analysis. The parasite module and */proc/kcore* approaches to memory acquisition both rely on methods that avoid the creation of kernel modules specific to each kernel version. While these techniques allow for acquisition from a wide range of systems, they also push the kernel version issue into the analysis phase.

In order for memory analysis tools to process Linux memory images, they must know the address of key symbols as well as the layout of data structures. This is also the exact information needed to create kernel modules that can load against a running system. By skipping the gathering of this information during acquisition, the work is passed onto the investigator performing analysis. If the investigator cannot create a profile, then he must fall back to unstructured analysis (strings, grep, etc.) and forego the true power of memory forensics.

Android acquisition. Attempting to acquire memory from Android devices entails all of the previously described problems with Linux acquisition along with several additional ones.

The issues facing Android memory acquisition include:

- The inability to bypass locked screens
- The need to root the device without rebooting the phone
- The need to gather the kernel headers and configuration for the phone, for which vendors are incredibly slow to produce and often never produce at all
- The lack of */proc/kcore* functionality, as most of the millions of Android devices are on the 32-bit ARM platform
- The optional presence of */dev/mem*, and in the authors' testing, attempting to read from */dev/mem* on Android devices always resets the phone. A valid memory capture has never been obtained in this manner, in our experience.

These problems are described in even greater detail in Wächter and Gruhn (2015) where the researchers attempted to acquire memory from common Android devices in their stock configuration. These tests mimicked scenarios encountered in the wild, and the paper details each step taken and where the procedures failed or succeeded. The community has struggled with these problems with Android analysis for several years, but no solution has been discovered (masdif, 2014).

Future directions

The use of the parasite method and of */proc/kcore* simply push the issue of kernel version independence on the analysis tool and forensic analyst. Linux memory analysis tools are highly dependent on per-version information and no research effort to date has successfully made them less dependent across a wide variety of systems. We feel that the most robust solution to this problem will be a database of information about a wide range of systems that is kept up-to-date, publicly available, and seamlessly supports incident response in unknown environments. This system would support the needs of both acquisition and analysis tools, but will require partners across industry in order for such an effort to be sustainable.

For Android analysis, the reliance on full physical memory dumps may need to be reevaluated due to the difficulty in obtaining them. As discussed in Ali-Gombe (2012), there is great value in even userland memory dumps that contain all of the data accessible to a process. Particularly for law enforcement investigations, such data will include passwords, emails, browsing history, chat messages, call logs, and more. Userland analysis will put malware investigators at a disadvantage, however, as any type of sophisticated userland or kernel-level malware will not be detected by such approaches.

Area of focus – memory analysis

Historical approaches to memory analysis

Memory analysis started in the early 2000s when digital forensic investigators realized that they could acquire memory directly from a running system through previously available interfaces. At the time, there were both rootkits that were difficult or impossible to detect on a running system as well as anti-forensics techniques that could fool live analysis tools. At this stage in the history of memory forensics, however, there were no mature frameworks for performing memory analysis, and instead, investigators had to rely on tools, such as strings, grep, and hex editors in order to find data of interest. This technique is now referred to as 'unstructured analysis' as it simply treats a memory capture as a raw stream of bytes. Unstructured analysis still has its uses, such as when searching for strings generated by user activity or for passwords and encryption keys, but thorough investigations require structured analysis.

In early 2005, DFRWS released their annual challenge. This challenge required investigators to perform thorough analysis of a Windows memory sample. This led to the creation of several memory analysis tools, including KntTools (Garner, 2005), MoonSols (Suiche, 2007), the FATKit (Petroni et al., 2006), VolaTools (Walters and Petroni, 2007), and Volatility (The volatility framework, 2016). In the years since, several powerful open source frameworks as well as commercial analysis tools have been developed. There have also been numerous academic papers and industry conference presentations that extend memory forensic capabilities related to malware detection, defeating anti-forensics, tracking user and attacker activity, and more.

In the following sections we discuss where these efforts have taken memory forensics investigators and where further work is needed to ensure that memory analysis remains an integral part of digital forensics and incident response processes.

Current issue – userland platform analysis

Current state

The threat of rootkits and the inability to detect them on live systems has led to a substantial amount of memory forensics research time being geared towards detecting system state anomalies in kernel memory. At the same time as this research was taking place, operating system vendors also took action against kernel rootkits. To start, both Microsoft and Apple decided to enforce that all kernel drivers must be signed (Case and Richard, 2016). This raises the bar for malicious actors as they must steal code-signing certificates from legitimate entities in order to get their driver loaded. While there are examples of suspected nation-state actors and criminal groups performing this task (http, 2016b; https, 2016c; http, 2016a) this is not

Please cite this article in press as: Case, A., Richard, III, G.G., Memory forensics: The path forward, Digital Investigation (2016), http://dx.doi.org/10.1016/j.diin.2016.12.004

something that less skilled attackers will be capable of accomplishing.

Microsoft also went a step further than Apple and implemented Kernel Patch Protection, which is commonly referred to as Patch Guard (Kernel patch protection, 2016). This protection mechanism operates from the kernel and protects kernel code from being modified as well as protects key data structures from being tampered with, such as the process list, callback handlers, and driver objects (Ionescu, 2015).

The rise of userland malware. The inability for malware authors to easily load their rootkits into the kernel has led to a surge in userland-based malware. Malware written to run in process memory has many of the same capabilities as kernel level malware, but is easier to write and does not require its code to be signed. Even without kernel access, userland malware can still spy on all of a user's activities, such as logging keystrokes, monitoring web cameras and microphones, stealing browser history, contact lists, chat conversations, and more and then exfiltrating stolen data over the network. Userland malware can also be extremely hard to detect on the live system as it has the access necessary to hook the APIs that live tools rely on in order to gather a view of a system's runtime state.

Current capabilities. The threat of userland malware necessitates that memory forensic frameworks provide robust detection capabilities against the malicious activity. Currently, these tools have strong capabilities in detecting traditional userland malware techniques, such as code injection, code modifications (inline hooks, API hooks, etc.) and manipulation of the runtime loader, such as IAT/EAT patching on Windows, GOT/PLT patching on Linux, and so on (Ligh et al., 2014). There is also strong support on Windows for abuse of APIs through the native Windows GUI APIs (Omfw, 2012).

Issues and limitations

While memory forensic frameworks do provide the previously listed capabilities against userland malware, there is still much more work left to be done. In particular, memory forensic does not provide deep coverage of the many userland runtime platforms provided by each operating system in order to make development simpler and more standardized. Unfortunately, these platforms are already being abused by malware in order to maintain functionality without leveraging the operating system components traditionally checked by security agents and memory forensics tools. This leads to a lack of detection capabilities as the malware is operating higher in the application stack than the memory analysis algorithms know how to inspect. For the remainder of this section, we will discuss each of these platforms, along with the existing state of research, if any.

Windows. Memory forensics is currently lacking in two key areas for Windows systems. The first is the ability to detect Powershell activity in a post mortem investigation. As discussed in a recent Symantec report, Powershell is a key component of modern attacks, particularly those by sophisticated actors (The increased use of powershell in attacks, 2016). Beyond private toolsets, there are also open source projects, such as Powershell Empire (Powershell empire, 2016), that allow for pure Powershell post-exploitation control of systems. Empire performs all of its Powershell work completely in-memory, without actually executing `powershell.exe` on disk, and supports keylogging, credential theft, lateral movement, and more. As the prevalence of Powershell grows, the memory forensics community must respond to its capabilities, particularly given how many of the malicious scripts reside only in memory. Current capabilities against Powershell generally rely on string searching or looking for side-effects of the actions taken by Powershell scripts and not the actual Powershell activity itself.

Another important area in which further research is required is post-mortem examination of the .NET runtime inside process address spaces. .NET malware has already been found in numerous real world attacks and malicious campaigns (SecureList, 2015; Cisco, 2014; MalwareBytes, 2016), but there are currently no memory forensic capabilities to detect such activity. Besides the malware found in the wild that abused .NET, there are also several open research projects that document the malicious activity. For example, the NTCore project has a detailed article on how code hooking and filtering can be performed at runtime (NTCore, 2016). .NET also provides runtime function overriding. Function overriding allows for malicious code to substitute the legitimate handler of functions calls with a malware-controlled implementation. This provides for a wide range of abuse as all APIs can be filtered, monitored, or completely replaced. There are currently no memory forensic capabilities that specifically check for .NET runtime manipulation by malware.

Mac OS X. Apple provides two runtimes to allow developers simple and standard access to a wide range of system resources, including memory management, hardware devices, user and GUI activity, and more. The first of these runtimes, Objective-C, is widely used throughout the operating system, and is heavily targeted by malware. As discussed in a recent DFRWS paper (Case and Richard, 2016), the notorious Crisis malware abused Objective-C in order to monitor user's browser activity, log system wide keystrokes, activate the microphone and webcam, and to hide from live analysis. The main method that Crisis employed was runtime method swizzling, which is the ability for code to replace method implementations at runtime. Crisis abused this feature to allow its own malicious handlers to spy on and manipulate data that was being processed by applications.

The previously referenced research provides the ability to detect Objective-C abuses, but was heavily targeted towards the 10.9 and 10.10 versions of OS X. Furthermore, this work is heavily dependent on the version of Objective-C being analyzed, much like memory analysis tools are dependent on versions of the operating system. In order for this work to become widely usable throughout the field, it must be extended to more versions of OS X.

The second Apple-provided runtime platform is Swift (Apple, 2016). This platform is meant to replace Objective-C in the future, but they are currently implemented in parallel. Unlike Objective-C, there are currently no memory forensics capabilities targeted towards Swift, leaving malware free to operate undetected.

Linux. While Linux is generally used from the command line, such as through SSH, there are specific engineering fields and companies that rely heavily on the Linux desktop. This is particularly true for the many organizations that leverage high performance applications for processing tasks related to video editing, engineering, and statistics. Crisis, along with several other malware samples found the wild, target the Linux graphical runtime, XOrg, to spy on user activity. As with Swift and .NET, there is currently no memory forensic capability to examine or detect this activity.

Android. Android applications, such as those downloaded from the Play Store, are powered by the Android runtime. On older versions of Android, this runtime was Dalvik, and provided a Java-like interface to develop applications. Memory forensics researchers implemented deep memory analysis capabilities against Dalvik (Case; Macht, 2012). These capabilities included recovery of all

Please cite this article in press as: Case, A., Richard, III, G.G., Memory forensics: The path forward, Digital Investigation (2016), http://dx.doi.org/10.1016/j.diin.2016.12.004

loaded runtime classes, the name, type, and value of all static and class variables of each class instance, and the handler address of class methods. Leveraging this data, investigators were quickly able to deal with packed malware, as the unpacked values would be in memory, as well as quickly being able to focus on key features of the malware. The main downside to this research is that it was difficult to use across a wide range of phones due to the data structures being highly dependent on the Dalvik version as well as general Android acquisition issues as described previously.

The research performed against Dalvik, while novel, has quickly becoming outdated as Dalvik is being replaced with a new Android Runtime (ART) (Android, 2016). ART changes the Dalvik method of JIT-based compilation with full compilation of applications as they are installed on the device. This replaces the generally easy to reverse engineer Dalvik classes with native code compiled to ELF binaries. It also considerably changes how memory forensic analysis would approach the platform, and to date there has been no public memory analysis research against ART.

Future directions

The increasing prevalence and power of userland malware requires attention by the memory forensics community. The runtime platforms that are implemented across operating systems are fertile grounds for malware as the platforms have complete control over the runtime and there are few security tools that adequately check for malware tampering.

We advise that memory forensics research projects that target these platforms do so in a manner that will discover a range of malware tampering and also support a wide range of OS versions. This process can be time consuming and difficult, however, as many of the platforms are fully closed source or at least partially closed source, which requires an experienced reverse engineer to perform the initial analysis.

Current issue – application specific analysis

Current state

Memory forensics was traditionally used only during malware analysis or in incident response when malware or advanced attackers were present. This has quickly changed over the last several years as investigators have realized the value of memory forensics during all types of investigations. This includes investigations targeting rogue insiders, anti-forensic applications, and during civil and criminal proceedings involving electronic devices. In many of these cases, memory forensics techniques are able to recover information that is not available to network or disk forensics. This occurs with browsers that implement 'private browsing', wherein many common artifacts are not written to disk, chat applications that implement end-to-end encryption and disable on-disk logging, such as Pidgin and Adium, and volume and file encryption tools that encrypt all information written to local storage. Memory forensics can assist in these situations since application information that is written in encrypted form to the local disk or the network is stored in cleartext while in memory for processing.

While highly useful, much of this information recovery is left to manual analysis, such as through examination of strings, by the investigator. We propose that to help scale investigations and to ensure that all artifacts are recovered, that memory forensic frameworks begin to deeply analyze application artifacts in a structured manner.

Issues and limitations

Memory analysis frameworks currently implement few application-specific analysis plugins. The Volatility framework has the most, but only implements a handful of these, such as plugins that recover information from Notepad, Notes, Pidgin/Adium, and the Calendar application. While these are all useful, they represent a small percentage of applications that hold forensic value. In the following sections we describe other applications and services for which investigations would greatly benefit from structured analysis.

Web browsers and browser activity. Although disk forensics of browser activity can often recover detailed information about a user's previous browsing activity, it often still leaves much to be desired. Particularly with the rise of HTTPS across websites, which prevents caching of files to disk, investigators are often left with only partial pictures of a user's historical behavior. For instance, when tracking data exfiltration, an investigator may be able to see that a person browsed to Dropbox or accessed his Gmail account, but due to HTTPS, will be unable to determine what data was actually transferred when relying solely on disk forensics. With memory forensics this changes though as fragments of the session will still be in memory. In real-world cases this has led to recovery of the name and full path of files recovered, and in the case of smaller files, complete recovery of the file itself.

A similar pain point is determining what data a user sent to web applications. Inside corporations, internal applications maintain highly sensitive information about employees and the company itself. These applications are a prime target for malicious insiders, but unfortunately web servers rarely store data sent through the POST HTTP method. Since all sensitive data is sent through this method, as opposed to GET, which makes data appear in the URL, important information is missed by investigators. With memory forensics techniques, this data can often be recovered and associated with specific web browser processes and user accounts.

This type of analysis currently requires an investigator to be highly familiar with HTTP and the furthermore, the investigator must manually analyze available evidence using strings and grep output to piece things together. The addition of automatic extraction of this information would be of tremendous value to investigators.

Office applications. Microsoft Office is installed on nearly every corporate end-user system, but there is currently no memory forensic support specific to this suite of applications. Analysis of Office activity is relevant to both insider threat investigations as well as malware investigations stemming from attacker's use of malicious documents. On the insider threat side, being able to determine the contents of documents viewed by users would be of tremendous value. Many times in these scenarios the investigator has proof that a user viewed a specific document, based on its file path recovered from LNK files, jumplist databases, and the registry, but the viewed file has since been deleted. This often stems from rogue insiders who download many documents from across the network, view them to look for relevant information, exfiltrate them if interesting, and then delete them locally afterwards. In order to definitively prove which documents a user was viewing, the investigator will often attempt to tie back file contents to files across the network. While this can be done through deleted file recovery, this process can be extremely time consuming given how many Office documents will appear on the storage devices of an average corporate user. Leveraging memory forensics features specific to Office analysis could greatly enhance this process.

On the malware side, the use of malicious macros and VBA scripts has quickly returned to real world attacks (Volexity, 2016). Sending malicious documents through email or other means is a quick way for attackers to target specific users and the general lack of security conscious behavior by non-technical end users means that the attacks have a high chance of succeeding. Memory

Please cite this article in press as: Case, A., Richard, III, G.G., Memory forensics: The path forward, Digital Investigation (2016), http://dx.doi.org/10.1016/j.diin.2016.12.004

forensics is currently only able to recover side effect activity of malicious documents, such as if the payload injects code into another process, but is not able to detect the malicious documents themselves.

Web servers. Investigations of attacks through web servers and services are often very frustrating. The only user-controlled data that web servers are capable of logging is the data passed through GET requests, which is data visible in the URL. As mentioned previously, any sensitive data is instead passed through POST, which is not logged by default and usually not logged at all. This leaves investigators in frustrating situations in which they are attempting to determine when attackers initially gained access to the system or what commands were sent to a web-based backdoor, but none of the relevant data is logged.

Memory forensics is often useful in these situations as the remnants of data from the malicious interactions will often still be stored in memory. This is true even of the unencrypted buffers of HTTPS connections that would be fully encrypted on the network. Again, however, memory forensic frameworks currently leave recovery of such data to the investigator and do not attempt to automatically extract such records even though they can be of high value.

Database servers. Investigations involving access to sensitive databases by attackers often face the same roadblocks as those associated with investigating web browsers and web servers. Database servers only log diagnostic and authentication information to disk, which leads to an incomplete picture of activity. Memory analysis can recover a plethora of data, including queries executed, results returned by queries, commands executed through functionality such as *xp_cmdshell* of MSSQL, and more. The difficulty is that with current generation tools, this work must be performed through unstructured analysis. With further research this data could be automatically extracted. For example, with MSSQL views, which are memory-only tables, a wide range of information, such as which user accounts authenticated, which queries were run by which user, as well as a full list of all queries executed against the running database are available (Fowler, 2007). Memory forensics frameworks that implement recovery of this information in a structured manner will provide rapid recovery of highly actionable artifacts.

Future directions

For memory forensics to be useful across all types of investigations and investigative scenarios, research must be performed that targets specific userland applications and the data they process. Current approaches to analysis of such data do not scale and are not easily repeatable, as they rely on a mix of manual extraction and examination of strings and the knowledge and skills of the investigator.

Current issue – compressed, in-memory swap

Current state

The importance of the integration of swap data into memory analysis has been discussed several times throughout this paper. In those instances, the discussion focused on the integration of swap from disk, but in modern operating system versions there is also a separate store of swap in memory. First discussed in a forensics context in Case and Richard (2014), these in-memory stores of swapped pages are highly compressed and provide a substantial performance improvement versus pages being written out to disk.

Issues and limitations

Swapped pages being in memory alleviates the issue of pagefile collection as discussed previously, but also introduces additional complications. To start, data being compressed in-memory means that it will not be discoverable by unstructured methods such as strings, grep, and Yara scans. These methods are still heavily relied upon by investigators to recover information that cannot automatically be recovered in a structured manner.

The other issue is that no memory forensics framework currently implements transparent handling of compressed swap pages. The referenced research implemented decompression capabilities for certain Linux and Mac Volatility plugins, but this only applied to particular portions of the framework. With the ongoing development of Volatility 3, and the easy incorporation of secondary memory sources, the integration of these swap sources should occur, but in the meantime investigations are missing artifacts from the compressed stores. As the success of these stores was realized on Mac and Linux, Microsoft decided to implement the functionality starting in Windows 10 (WindowsITPro, 2016). There is currently no research that documents the algorithms used by this compressed store, which means that the compressed pages are essentially inaccessible to memory forensics investigators.

Future directions

The heavily reliance of OS X, and now Windows, on in-memory compressed stores means that memory forensics frameworks must incorporate them into their page translation algorithms. Just as these frameworks must be adapted to read page files from disk in a transparent manner, they must also be able to recognize when pages are compressed and then decompress them transparently for analysis. Only then will the full range of information stored in memory become available. This will also allow traditional unstructured analysis to become fully usable again as the framework can extract and present the decompressed pages to the investigator.

Current issue – Windows 10

Current state

Windows 10 not only changes approaches related to memory acquisition, but also has effects on memory analysis. Based on current research, these include the introduction of native Linux support as well as changes to operating system update cycles.

Issues and limitations

Native Linux

Starting with Windows 10, Microsoft introduced a native capability to run Linux applications inside of Windows (MSDN, 2016b). There is currently no forensics-oriented research on this new feature, but noted security researcher Alex Ionescu has presented extensive research on this topic. His work was mostly focused on the architecture itself and how it appears on the live system, but this a foundation on which future forensics research could be performed.

Rapid kernel updates

Microsoft traditionally saved major updates to the operating system for service packs. This model has changed though in order for Microsoft to more quickly respond to security issues as well as adding features. This new approach to updates affects Windows 7, 8, and 10, but most heavily affects Windows 10. The rapid development of changes, sometimes in less than a week between releases, means that memory analysis tools must be constantly

Please cite this article in press as: Case, A., Richard, III, G.G., Memory forensics: The path forward, Digital Investigation (2016), http://dx.doi.org/10.1016/j.diin.2016.12.004

checked against the latest versions of the OS in order to detect major changes to data structures and algorithms.

Future directions

Native Linux

The introduction of Linux capabilities into Windows will require substantial forensics research. Ionescu has done an excellent job of documenting the current implementation, but future research faces a few hurdles. The first is that reverse engineering of a large subsystem is required for any substantial effort. Ionescu is one of the best Windows reverse engineers in the industry, which means his research efficiency is difficult to widely replicate. Furthermore, his research notes state that the Linux implementation has seen substantial changes since its introduction, which means that any research efforts may become quickly outdated until the interfaces and subsystem becomes more stabilized.

Rapid kernel updates

The introduction of rapid kernel updates changes the way that memory forensics tools traditionally represented the data structures and algorithms of analyzed operating systems. For example, Volatility 2.x distributes Windows profiles on service pack boundaries, such as a profile for Windows 7 service pack 1 or Windows XP service pack 2. This model is now inadequate as major changes to key data structures occur in minor releases of the operating system. This has led to the Volatility developers needing to generate new profiles for the subset of updates that change key data structures. This also requires the end user to pick specific sub-versions of an operating system to obtain complete analysis.

Other memory analysis tools, such as WinDBG (MSDN, 2016c), Rekall (Google, 2016), and the future Volatility 3 work around this issue by incorporating debug symbols (PDB files) into analysis. Incorporation of PDB files fixes the issue related to data structure mismatches between subversions, but does not automatically fix issues when algorithms change between versions. Ongoing work with Windows 10 shows that these changes occur frequently and require manual updates to the processing algorithms of memory analysis frameworks.

Sole reliance on PDBs also introduces practical problems with memory analysis as not all modules have PDB files, such as the network stack (tcpip.sys) and the GUI subsystem (win32k.sys). These means that manual reverse engineering must still be done after updates that break existing analysis plugins. The use of PDBs is also cumbersome to investigators who must download and maintain a set of PDBs for all analyzed systems. This is especially difficult for investigators who work in closed environments, which is prevalent in government organizations and also many private forensics firms.

Technologies without memory forensics coverage

So far, we have discussed platforms and operating systems for which memory forensics capabilities exist, but which require further work. We conclude the paper by discussing other important systems for which memory forensics is sorely needed, but for which no physical memory analysis capabilities exist.

Apple iOS

After Android, Apple's iOS, which powers iPhones and iPads, has the largest market share in the world. Due to its popularity and security features, it is often the platform of choice for politicians, businessmen, dignitaries, diplomats, and other people who hold very sensitive information. This has led to several samples of malware that targeted the platform (Lookout, 2016; theiphonewiki, 2016) in order to spy on user activity.

To date, there has been no public memory forensics research published for iOS devices. The largest stumbling block to such research is the inability to directly acquire physical memory. Even on jailbroken devices, users cannot directly load kernel modules as the facilities are not present in the operating system. This prevents a tool, such as LiME, from being written for Apple's iOS. It is also in contrast to Android where root users are able to directly load kernel modules.

Gaining kernel access to iOS devices is not impossible, as illustrated by the number of available kernel exploits. In fact, Stefan Essar, one of the most noted members of the jailbreak community, developed a hardware kernel debugger for older versions of the iPhone (Esser, 2011). Leveraging exploits to enter the kernel and later get a kernel module loaded seems like a plausible avenue for memory forensics. It would still be highly dependent on the phone in question though as the phone would either have to already be jailbroken or be vulnerable to exploits that allow for triggering of a kernel vulnerability without a reboot. Furthermore, it's very likely that any vulnerabilities exploited to allow memory acquisition will be quickly patched.

Chromebooks

Google's Chromebooks are locked down computing “appliances” that run a heavily customized version of Linux. Chromebooks provide no native access to the file system or to memory. For security reasons, users are shielded from all parts of the operating system that digital forensics analysts traditionally collect for analysis. As with iOS, unless there is an exploit that allows for bypassing of the built-in security measures, then full forensics collection is not possible.

The Internet of Things

The billions of devices that power the “Internet of Things” (IOT), are increasingly being used in situations that require forensic analysis. These include devices that track information related to civil matters as well as traditional criminal activity on top of recent major botnet attacks (Schneier, 2016; ThreatPost, 2016). Many of these devices are built upon Linux, and due to their restrictive access environments and lackluster vendor support for 3rd party kernel code, it is difficult or impossible to acquire memory samples. Furthermore, devices running custom operating systems and hardware have no memory forensics support at all. As these devices become more and more common, the forensics community must perform research to support a wide range of devices, all of which run different operating system versions, hardware architectures, and software configurations. The one thing in common with these devices is the goal of preventing outside access. It seems likely that any approach to memory acquisition of IOT devices will require chaining acquisition tools with exploits. Adoption of such an approach will have raise immediate legal questions about admissibility as well as technical questions to ensure stable acquisition. As with iOS devices, any vulnerabilities that can be leveraged to provide memory acquisition will likely be patched.

Closing thoughts

Memory forensics has proven to be one of the most versatile and powerful ways to analyze computer systems. It has become a daily part of incident response procedures as well as the driving force behind proactive analysis of environments for malicious activity. In this paper we discussed historical work that was performed to

Please cite this article in press as: Case, A., Richard, III, G.G., Memory forensics: The path forward, Digital Investigation (2016), http://dx.doi.org/10.1016/j.diin.2016.12.004

enable the current power of memory forensics as well as improvements that should be made in order to ensure that memory forensics stays at the forefront of defensive technology.

References

Ali-Gombe, A.I., 2012. Volatile Memory Message Carving: a “Per Process Basis” Approach.

Amazon EC2 Container Service Developer Guide, 2016. http://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html.

Android, 2016. Art and Dalvik. https://source.android.com/devices/tech/dalvik/.

Apple, 2016. Swift Developer Documentation. https://developer.apple.com/swift/.

Burdach, M., 2006. Physical Memory Forensics. https://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Burdach.pdf.

Carvey, H., 2005. Page Smear. http://seclists.org/incidents/2005/Jun/22.

Case, A., Forensic memory analysis of android's dalvik vm, in: Source Seattle.

Case, A., 2012. Phalanx 2 Revealed Using Volatility. https://volatility-labs.blogspot.com/2012/10/phalanx-2-revealed-using-volatility-to.html.

Case, A., Richard III, G.G., 2014. In lieu of swap: analyzing compressed ram in mac os x and linux. In: Proceedings of the 14th Annual Digital Forensics Research Workshop (DFRWS 2014).

Case, A., Richard III, G.G., 2016. Detecting objective-c malware through memory forensics. In: Proceedings of the 16th Annual Digital Forensics Research Workshop (DFRWS 2016).

Cisco, 2014. Reversing multilayer.net Malware. https://blogs.cisco.com/security/talos/reversing-multilayer-net-malware.

CrazyLord, 2002. Playing with windows/dev/(k)mem. http://www.phrack.org/archives/issues/59/16.txt.

Device guard deployment guide, 2016. https://technet.microsoft.com/en-us/itpro/windows/keep-secure/device-guard-deployment-guide.

Esser, S., 2011. Targeting the Ios Kernel. http://www.slideshare.net/i0n1c/syscan-singapore-2011-stefan-esser-targeting-the-ios-kernel.

Farmer, D., Venema, W., 2005. Forensic Discovery, vol. 6. Addison-Wesley.

Fowler, K., 2007. Sql Server Database Forensics. https://www.blackhat.com/presentations/bh-usa-07/Fowler/Presentation/bh-usa-07-fowler.pdf.

Garner, G., 2005. Knt Tools. http://www.gmgsystemsinc/knttools.

Garner, G., 2016. Threat Protection for Linux. https://www.forcepoint.com/product/security-cloud/threat-protection-linux.

Geek, H., 2015. What Is swapfile.sys and How do You Delete it? http://www.howtogeek.com/225143/what-is-swapfile.sys-and-how-do-you-delete-it/.

Google, 2016. Rekall. https://github.com/google/rekall.

http://arstechnica.com/security/2016/03/to-bypass-code-signing-checks-malware-gang-steals-lots-of-certificates/, 2016.

http://www.pcworld.com/article/3048417/malware-authors-quickly-adopt-sha-2-through-stolen-code-signing-certificates.html, 2016.

https://www.symantec.com/connect/blogs/suckfly-revealing-secret-life-your-code-signing-certificates, 2016.

Ionescu, A., 2015. What are little patchguards made of? http://www.alex-ionescu.com/?p=290.

Kernel patch protection, 2016. https://en.wikipedia.org/wiki/Kernel_Patch_Protection.

Ligh, M., 2012. Movp 3.1 Detecting Malware Hooks in the Windows gui Subsystem. https://volatility-labs.blogspot.com/2012/09/movp-31-detecting-malware-hooks-in.html.

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

Lookout, 2016. Sophisticated, Persistent Mobile Attack against High-value Targets on Ios. https://blog.lookout.com/blog/2016/08/25/trident-pegasus/.

Macht, H., 2012. Dalvikvm Support for Volatility. http://lists.volatilesystems.com/pipermail/vol-dev/2012-October/000187.html.

MalwareBytes, 2016. Unpacking yet another.net Crypter. https://blog.malwarebytes.com/threat-analysis/2016/07/unpacking-yet-another-net-crypter/.

masdif, 2014. Lime in Real World Android Forensics. http://lists.volatilesystems.com/pipermail/vol-users/2014-May/001254.html.

MSDN, 2016. Intro to the Universal Windows Platform. https://msdn.microsoft.com/en-us/windows/uwp/get-started/universal-application-platform-guide.

MSDN, 2016. Bash on Ubuntu on Windows. https://msdn.microsoft.com/en-us/commandline/wsl/about.

MSDN, 2016. Download the Wdk, Windbg, and Associated Tools. https://developer.microsoft.com/en-us/windows/hardware/windows-driver-kit.

niekt0, fmem, 2011. http://hysteria.cz/niekt0/.

NTCore,.net Internals and Code Injection, 2016. http://www.ntcore.com/files/netint_injection.htm.

Omfw, 2012. Malware in the Windows gui Subsystem. https://volatility-labs.blogspot.com/2012/10/omfw-2012-malware-in-windows-gui.html.

Petroni, N., Walters, A., Fraser, T., Arbaugh, W., 2006. Fatkit: a framework for the extraction and analysis of digital forensic data from volatile system memory. Digit. Investig. 3.

Powershell empire, 2016. https://www.powershellempire.com.

Rutkowska, J., Tereshkin, A., 2008. Bluepilling the xen hypervisor. Black Hat USA.

Scaling Based on CPU or Load Balancing serving capacity, 2016. https://cloud.google.com/compute/docs/autoscaler/scaling-cpu-load-balancing.

Schatz, B., 2007. Bodysnatcher: towards reliable volatile memory acquisition by software. Digit. Investig. 4, 126–134.

Schneier, B., 2016. Ddos Attacks Against dyn. https://www.schneier.com/blog/archives/2016/10/ddos_attacks_ag.html.

SecureList, 2015. The Rise of.net and Powershell Malware. https://securelist.com/blog/research/72417/the-rise-of-net-and-powershell-malware/.

Stüttgen, J., Cohen, M., 2014. Robust linux memory acquisition with minimal target impact. Digit. Investig. 11, S112–S119.

Suiche, M., 2007. Moonsols.

Suiche, M., 2008. Windows Hibernation File for Fun nprofit. Black Hat.

Sylve, J., Case, A., Marziale, L., Richard III, G.G., 2012. Acquisition and analysis of volatile memory from android devices. Digit. Investig. 8.

Sylve, J., Marziale, L., Richard III, G.G., 2016. Modern windows hibernation file analysis. Digit. Investig. (in press).

The increased use of powershell in attacks, 2016. https://www.overleaf.com/6919029ggkynfmkjvss#/23651472/.

The Volatility Framework: Volatile Memory Artifact extraction Utility Framework, 2016. https://github.com/volatilityfoundation/volatility.

theiphonewiki, 2016. Malware for ios. https://www.theiphonewiki.com/wiki/Malware_for_iOS#Tools_used_by_governments_.28and_similar.29_to_target_individuals.

ThreatPost, 2016. Mirai-fueled iot Botnet Behind ddos Attacks on dns Providers. https://threatpost.com/mirai-fueled-iot-botnet-behind-ddos-attacks-on-dns-providers/121475/.

Volexity, 2016. Powerduke: Widespread Post-election Spear Phishing Campaigns Targeting Think Tanks and ngos. https://www.volexity.com/blog/2016/11/09/powerduke-post-election-spear-phishing-campaigns-targeting-think-tanks-and-ngos/.

Wächter, P., Gruhn, M., Practicability study of android volatile memory forensic research, in: Information Forensics and Security (WIFS), 2015 IEEE International Workshop on, IEEE, pp. 1–6.

Walters, A., Petroni, N., 2007. Volatools. https://www.blackhat.com/presentations/bh-dc-07/Walters/Paper/bh-dc-07-Walters-WP.pdf.

WindowsITPro, 2016. Understanding Compressed Memory in Windows 10 Anniversary Edition. http://windowsitpro.com/windows-10/understanding-compressed-memory-windows-10-anniversary-edition.

Please cite this article in press as: Case, A., Richard, III, G.G., Memory forensics: The path forward, Digital Investigation (2016), http://dx.doi.org/10.1016/j.diin.2016.12.004