Memory Analysis of macOS Page Queues

By:
Andrew Case (Volatility Foundation), Modhuparna Manna (Louisiana State University), Ryan Maggio (Louisiana State University), and Golden Richard (Louisiana State University)

From the proceedings of

The Digital Forensic Research Conference

DFRWS USA 2020

July 20 - 24, 2020

Forensic Science International: Digital Investigation 33 (2020) 301004

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

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

DFRWS 2020 USA — Proceedings of the Twentieth Annual DFRWS USA

Memory Analysis of macOS Page Queues

Andrew Case [a], Ryan D. Maggio [b, c], Modhuparna Manna [b, c], Golden G. Richard III [b, c, d, *]

[a] *Volatility Foundation, USA*
[b] *Center for Computation and Technology, Louisiana State University, USA*
[c] *School of Electrical Engineering & Computer Science, Louisiana State University, USA*
[d] *Department of Computer and Information Sciences, Towson University, USA*

ARTICLE INFO

Article history:

Keywords:
Memory forensics
Incident response
Digital forensics
Operating systems

ABSTRACT

Memory forensics is the examination of volatile memory (RAM) for artifacts related to a digital investigation. Memory forensics has become mainstream in recent years because it allows recovery of a wide variety of artifacts that are never written to the file system and are therefore not available when performing traditional filesystem forensics. To analyze memory samples, an investigator can use one of several available memory analysis frameworks, which are responsible for parsing and presenting the raw data in a meaningful way. A core task of these frameworks is the discovery and reordering of non-contiguous physical pages in a memory sample into the ordered virtual address spaces used by the operating system and running processes to organize their code and data. Commonly referred to as address translation, this task requires a thorough understanding of the memory management mechanisms of the hardware architecture and operating system version of the device from which the memory sample was acquired. Given its critical role in memory analysis, there has been significant interest in studying the operating system mechanisms responsible for allocating and managing physical pages so that they can be accurately modeled by memory analysis frameworks. The more thoroughly the page handling mechanisms are modeled in memory forensics tools, the more pages can be scrutinized during memory analysis. This leads to more artifacts being reconstructed and made available to an investigator. In this paper, we present the results of our analysis of the macOS page queues subsystem. macOS tracks pages in a number of different states using a set of queues and as we will illustrate, the reconstruction of data from these queues allows a significant number of memory pages to be analyzed that are currently ignored by memory forensics tools. Through incorporation of these artifacts into analysis, memory analysis frameworks can present an even richer set of artifacts and data to investigators than ever before.

1. Introduction

The use of volatile memory analysis has become standard practice in the fields of digital forensics and incident response. Memory forensics allows investigators to analyze the entire state of a system, including kernel code and data, as well as the code and data of all running applications. Historical information can also be recovered through the discovery and examination of deallocated data structures. This crucial information is largely unavailable when applying traditional forensics techniques, particularly when malware is present and employing anti-forensics mechanisms. The rise of memory-only code and data in real-world malware and exploits has also contributed significantly to the popularity of memory analysis. This includes open source attack frameworks, such as Powershell Empire (PowerShell Empire, 2016) and Metasploit (hdm, 2020), commercial products such as Cobalt Strike (Mudge, 2020), and numerous exploits and malware samples found in the wild.

To perform memory analysis, investigators rely on memory forensics frameworks that translate the unordered set of physical pages in a memory sample into the structured and ordered virtual address spaces in which the operating system and all running applications operate. The process of associating virtual address spaces with their corresponding physical pages is known as address translation. Each hardware architecture defines the mechanism by which the hardware state describes whether a page in virtual memory is present in physical memory. Recovery of pages that are

* Corresponding author. Center for Computation and Technology, Louisiana State University, USA.
E-mail addresses: andrew@dfir.org (A. Case), rmaggi2@lsu.edu (R.D. Maggio), mmanna3@lsu.edu (M. Manna), golden@cct.lsu.edu (G.G. Richard).

https://doi.org/10.1016/j.fsidi.2020.301004

present in physical memory is very straightforward as the hardware state encodes the offset in the memory sample. For pages that are not marked as present in physical memory (i.e., "invalid" pages), the operating system defines the hardware state to encode the actual location of the page, such as in a paging file, a file on disk, in a compressed data store, or elsewhere. The ability to decipher these invalid states and recover the associated pages is a critical component of a memory forensics framework. We will illustrate later in the paper how relying solely on pages marked as present in physical memory leaves a substantial number of pages behind and negatively impacts results produced during analysis.

In this paper, we discuss our research effort to study the page queues of macOS to integrate analysis of the queue states into memory forensics. We showcase the usefulness of the queues by examining how prevalent the invalid page states are and how analyzing the queues makes a significant amount of currently inaccessible data available. To analyze these queues across a variety of operating system versions and memory samples, we implemented support into the Volatility framework (The Volatility Framework:, 2017). Our code not only provides statistical analysis of the queues, but also integrates analysis of the queues directly into Volatility. Through use of our developed code, all existing and future Volatility plugins automatically benefit from the queue analysis. We plan to submit our developed code into the public Volatility repository upon publication so that the entire memory forensics community benefits.

2. Related work

Given the crucial role that address translation plays in the memory forensics process, there have been many research efforts to decipher the operating system-specific encoding of invalid pages so that they can be located and incorporated into analysis.

In 2007, several papers were published that addressed a variety of these sources for Windows, including the recovery of invalid pages from paging files, transition pages, and prototype pages (Petroni et al., 2006; Walters and Petroni, 2007; Jesse, 2007). Paging files, also referred to as swap files, are used to store excess pages of memory that have not been recently or frequently accessed to free physical memory pages for more actively used data. Transition pages are those marked to be written to disk, but that are still currently in physical memory. They can simply be treated as valid pages by a memory forensics framework. Prototype pages are those shared between processes. Publications by Stimson in 2008 (Jared, 2008) and Iqbal in 2009 (Iqbal, 2009) deeply explored the incorporation of paging files into memory analysis.

In a blog post describing the incorporation of paging files into the Rekall memory forensics framework (Google, 2016), Cohen also describes how invalid pages that are stored within arbitrary, non-paging files in the filesystem can be recovered during address translation (Cohen, 2014). This is very useful, as the on-demand loading strategy of operating systems only loads in pages from files on disk into memory when they are accessed ("on demand") or when they are part of a read-ahead set for applications that access many portions of a file.

To evade Linux malware that maps regions with the PROT_NONE protection flag, Volatility specifically checks for pages marked as not present but that have the 8th bit in the page table entry set, which signifies a "global" page. The full explanation is given in a blog post (Levy, 2015), and the end result is that these pages are no longer hidden from Volatility even though they are not marked as present within the hardware state.

The addition of compressed paging stores has also driven considerable interest in the address translation mechanisms of operating systems. These stores are used to hold compressed pages inside of memory to avoid the performance penalties of writing them out to disk. The power of modern CPUs, combined with the performance of modern compression algorithms, means that a considerable number of pages can be efficiently compressed and decompressed from one physical page inside the store. Case and Richard analyzed these stores on Linux and Mac systems in a 2014 DFRWS paper (Case and Richard, 2014). Microsoft did not add support for compressed stores until Windows 10, and recently a team from FireEye presented their analysis of the Windows 10 stores. This included patches that enabled support for analyzing the stores inside of Volatility and Rekall.

Although our research effort is focused on an entirely unexplored method to recover additional pages during macOS address translation, we chose to cite these historical works as they showcase the power of accurately modeling address translation and inspired our effort.

3. Understanding macOS memory queues

Our effort began by studying the page fault handler of the macOS kernel to determine the states that individual pages might be in. Because many of the data structures used by the macOS virtual memory system might be unfamiliar to many readers, we suggest that readers who are committed to closely following the discussion below consult two important sources. The first is an authoritative text on macOS internals, which, while dated, remains very useful for understanding the intricacies of the virtual memory system (Amit, 2006). Of particular importance is Figure 8.6, which illustrates the relationships between many important data structures. The second is the kernel source itself, as there is very little additional documentation besides the kernel source. The macOS (xnu) kernel source code is made available by Apple on its website (Apple, 2020c) and Github (Apple, 2020a). We attempt to point out locations within the source where important data structures and functions are defined in the following.

The page fault handler is called when code inside the kernel or inside of a process tries to access a virtual address whose backing physical page is not currently present. This makes the page fault handler responsible for dealing with the situations mentioned previously, such as accessed pages being held in paging files, compressed stores, and other invalid states. Our goal was to determine which, if any, states were viable candidates for memory forensics tools to use even though the hardware state has them marked as non-present.

3.1. The page fault handler

The *vm_fault_internal* function (defined in vm_fault.c) performs the bulk of the operating system's page fault handling duties. *vm_fault_internal* begins by locating the memory map associated with the virtual address that caused the fault through use of the *vm_map_lookup_locked* (vm_map.c). *vm_map_lookup_locked* returns the *_vm_map* data structure (associated with the faulting address as well as the *vm_object* and *vm_object_offset* values.

The *_vm_map* data structure in macOS is the equivalent of the VAD structure in Windows (Dolan-Gavitt, 2007), and holds metadata about a memory region of process, such as the starting and ending address, permissions, and mapped file, if any. *vm_map* is defined in the vm_map.h header file in the kernel source. The *vm_object* data structure (vm_object.h) tracks the physical pages of a particular object and defines how to find the associated pages. For pages in physical memory, there is a list of pointers to *vm_page* data structures (defined in vm_page.h), one for each page, and for non-resident pages, the object tracks the *pager* responsible for gathering the pages. Pagers exist for data stored in locations such as the compressed store or files on disk. *vm_object_offset* is simply an integer value that holds a particular page's offset into its host object.

3.2. vm_page_lookup and vm_page_buckets

Once the map, object, and offset are found, the page fault handler then uses the *vm_page_lookup* (defined in vm_resident.c) function by passing it the object and offset to find the associated *vm_page*. *vm_page_lookup* first checks the per-object cache to locate the object/offset pair, and if that fails it then relies on the use of *vm_page_buckets*. The remainder of this section and paper will focus on *vm_page_buckets* as during memory analysis we often want to recover all the data associated with an object, not just that which happens to have been stored in a look-aside cache during acquisition.

vm_page_buckets is a hash table, defined in source file vm_resident.c, that maps object/offset pairs to their corresponding *vm_page* structures. This hash table has an entry for each physical page currently tracked by the operating system, and it plays a similar role to that of the page frame number (PFN) database of Windows (Karvandi, 2018) and the *mem_map* data structure of Linux (Gorman, 2014).

Through our analysis of the page fault handler, we discovered that by analyzing *vm_page_buckets*, a memory forensics framework can derive the physical offset of any virtual address, even when the virtual address is marked as not present. The only exceptions to this are when the virtual address truly has no backing physical page or when the page is in an error state.

3.3. vm_page and page states

In macOS, each physical page is associated with a *vm_page* structure, as mentioned above, which tracks its associated object and offset, its current state, its PFN, and a few other pieces of metadata. The physical offset of a page can be calculated by multiplying its PFN by 4096, which is the size of a hardware memory page on modern Intel processors. Before accessing a page, its state must first be examined. Scrutiny of *vm_ fault_internal*, as well as the *vm_ fault_page* helper function, illustrated that several problematic page states are skipped when the operating system attempts to recover a non-present page. The page states that are skipped are *unusual*, *error*, *busy*, *absent*, *fictitious*, and *restart*. These states and others are enumerated in the kernel source in vm_page.h (at line 245 in the Mojave source tree). For pages that are not in one or more of these problematic states, the particular memory queue that the page is on is then checked for suitability of recovery.

3.4. Memory queues and queue states

The queue that a page is currently on is tracked inside of its *vm_page* structure. The queues used by the macOS kernel to track pages are:

- **Wired**: pages that cannot be swapped. Similar to the non-paged pool of Windows.
- **Compressed**: pages that are being used by the compressed swap subsystem. They must be decompressed before use.
- **Free**: several queues hold free pages.
- **Throttled**: previously busy pages eligible to be made active.
- **Page Out**: pages prepared to be paged out.
- **Active**: pages active in memory.
- **Inactive External**: queue of anonymous (non-file backed) pages.
- **Inactive Internal**: queue of file-backed pages.
- **Inactive Cleaned**: pages that were previously dirty but have since been written to a backing store.
- **Speculative**: read-ahead pages from objects recently accessed.
- **Secluded**: memory reserved for the Camera application.

A set of macros and constants in vm_page.h (line 162 in the Mojave source tree) is used to associate pages with these queues. There is also a transient state of “not in a queue” for pages that were recently allocated but not yet used. By comparing a page's state to its current queue, the kernel can decide if its eligible to be retrieved during a page fault. Along with the previously mentioned macOS kernel functions, we also studied the *memory_object_lock_page* (memory_object.c) and *vm_pageout_scan* (vm_pageout.c) functions to determine the full relationship between pages, page states, and the page queues.

4. Memory analysis of the page queues

After studying the kernel source related to the page fault handler and the paging subsystem, we were able to develop a memory forensics algorithm that successfully recovers pages that are still in memory but marked as being non-present in the hardware page state. This algorithm involves several steps. First, for a given virtual address in an address space, it must be determined if there is a corresponding *vm_page* structure. If the corresponding structure exists, it must be checked for eligibility to be recovered based on its state and current memory queue. Finally, its physical offset must be extracted so that the corresponding data can be used during analysis.

4.1. Locating vm_page instances

The first step of the developed algorithm is to locate *vm_page* instances. We developed two methods for this, one to aid with testing and the other for integration into Volatility's macOS page fault handler.

4.1.1. Enumerating vm_page_buckets

To aid our understanding of the page queues and to provide statistical analysis, we first developed a Volatility plugin, *mac_walkqs*, that enumerates every page tracked by *vm_page_buckets* and reports each page's metadata. This plugin begins by creating an array of *vm_ page_bucket_t* typed elements and with an array size determined by the *vm_page_bucket_count* global variable. Each bucket of the array references a list of *vm_page* instances. For each instance eligible for recovery, the plugin prints the *vm_ page* address, its PFN, the address of the page's corresponding object, its offset in the object, and which queue the page is stored in. As discussed, understanding the distribution of these states helped us to understand what types of data we should expect to recover during structured analysis. Section 6 presents analysis of the states of each page for our test samples.

4.1.2. Indexing vm_page_buckets

The second Volatility plugin that we developed, *mac_find_ page*, replicates the macOS page fault handler's ability to locate *vm_page* instances for specific non-present virtual addresses in specific contexts. This plugin begins by locating the map data structure (*vm_map_entry*) instance corresponding to the given virtual address. This is accomplished by leveraging the Volatility *get_proc_maps* function. Next, the plugin discovers the object corresponding to the map and the map's offset into that object. Extracting the offset is straightforward as it is simply an integer field in the map data structure. Extracting the object requires several steps, however. The first step is to access the object structure. The pointer to this structure inside of the map is embedded inside a union, as a map can represent either an actual object or a *sub_map* (see vm_map.h). Our plugin checks for maps that are submaps by examining the *is_sub_map* member and bailing out if it is set. For actual objects, after getting the initial object structure, it must then be checked to see if it has a *shadow* object (Apple, 2020b). Shadow objects are created when an object is copied, such as during a process fork. Changes (writes) to the object are propagated to the shadow object by other objects that shadow it. To use the correct object structure, our code uses the shadow object of map objects when they are present.

After finding the map corresponding to a given virtual address and extracting the correct offset and object, our plugin is able to calculate an index into the *vm_page_ buckets* hash table. This is the same operation performed by the macOS page fault handler and its use of the *vm_ page_hash* macro, defined in vm_resident.c. By indexing the table directly, we can quickly narrow our search to just a single bucket list of *vm_page* instances. In our testing, these lists are generally very small, usually less than ten elements. To determine the matching *vm_ page* instance, we use the same algorithm as the page fault handler, which matches the extracted object and offset with those embedded in the *vm_page*. Through use of this algorithm, our plugin is able to quickly determine, which, if any, *vm_page* instance references our given virtual address and context.

4.2. Determining eligibility for recovery

Once a *vm_page* instance is found, we then must verify that the data at its referenced PFN is suitable for recovering. The suitability of an instance is first determined by examining its *unusual*, *error*, *absent*, and *fictitious* state members. Although the macOS handler also skips the *busy* and *restart* states, these are states our code can work around. As mentioned previously, if the page is in any of the error states then the corresponding physical page must be ignored. After examining its state, the queue on which the page resides must be examined. Our code considers pages in the *wired*, *throttled*, *page out*, *active*, *inactive internal*, *inactive external*, *internal cleaned*, *speculative*, and *secluded* states to be suitable for recovery. All of these reference pages that are still in memory, but that are marked as non-present in the particular address space being examined. We skip pages in the compressed state as previous work incorporated recovery of these (Case and Richard, 2014). We also skip pages on any the free queues as we do not currently believe that these pages are in a suitable state for recovery based on our reading of the kernel source code. We plan to revisit our assessment of each free queue as part of our future work effort.

4.3. Full integration into volatility

To allow all existing and future macOS Volatility plugins to benefit from our memory queue analysis algorithm, we needed to add the support inside of Volatility's page fault handler and not just as an individual plugin. Volatility implements per-architecture address translation in classes known as address spaces. There are currently implementations for AMD64, 32-bit Intel, 32-bit Intel with PAE, and 32-bit ARM. These classes implement an interface that supports translating virtual addresses to their physical offset, reading from virtual addresses, and more. Beyond the hardware-specific classes, there are also additional classes for Windows and Linux that add the operating-system specific address translation features discussed in Section 2. To begin our integration effort, we created an macOS-specific address space that inherited from the existing AMD64 class. We then implemented our own versions of the *vtop* and *get_paddr* functions.

vtop, which stands for *virtual to physical*, translates a virtual address to its physical offset. Our custom implementation first checks if the AMD64 address space considers it present. If so, we return this result. If the AMD64 address space considers the page non-present, we re-use our algorithm described in Section 4.1.2 to find the corresponding map, extract the object and offset, and leverage *vm_page_buckets*. The *get_paddr* function is also used to get the physical offset of a virtual address, and we provide a custom implementation that also leverages the memory queues to locate pages for virtual addresses the hardware state considers non-present.

For performance reasons, our address space uses a cache of the process mappings to avoid having to enumerate the maps on each translation. This cache stores the start and end address, actual object, and object offset for each memory region. The cache is generated on the first address translation and then re-used for all future translations inside a particular address space. Section 7 demonstrates the data recovered by our algorithm as well as the performance statistics.

5. Testing environment and data set

To test our algorithms for completeness, robustness, and reasonable performance, we generated a corpus of memory samples with diverse operating system version and hardware configurations. In this section we list the memory samples generated as well as the system used to test our Volatility integration.

5.1. Memory samples

Table 1 lists the memory samples generated for testing our integration. Using VMware, we created 2 GB and 4 GB samples covering every version of macOS from the latest Catalina all the way back to Mavericks, which is the same version coverage provided by Volatility itself. The use of small memory sizes was simply for expedience and to illustrate that pages are present in the queues for a variety of versions of macOS. Obviously, the number of pages for the 2 GB systems, which were booted and then immediately analyzed, will be small, but pages are still present in the queues. The 16 GB and 64 GB samples were taken from a MacBook Pro and a Mac Pro. These are the daily use systems of two members of the research team and were used to provide real world data sets.

Memory snapshots of the virtual machines were acquired using VMWare's built in facilities (Volatility Foundation, 2014). Memory snapshots of the physical systems were acquired using Surge Collect Pro (Volexity, 2020). All of the virtual machine systems were acquired within an hour of being booted. The MacBook Pro had an uptime of 41 days when acquired and the Mac Pro 30 days. If systems with 64gB and 128 GB had been recently rebooted, the number of pages present in the queues would be much smaller, mirroring the 2 GB VM cases more closely. The point is that pages

Table 1
Memory samples used for testing.

#	Name	Version	Build	Size (GB)
1	Mavericks	10.9.5	13F1712	2
2	Yosemite	10.10.5	14F1021	2
3	ElCapitan	10.11.4	15E65	2
4	Sierra	10.12.5	16F73	2
5	HighSierra	10.13.5	17F77	2
6	Mojave	10.14.5	18F132	2
7	Catalina	10.15.2	19C57	2
8	Mojave	10.14.4	18E226	4
9	Catalina	10.15.1	19B88	16
10	Catalina	10.15.2	19C57	64

are definitely present in the queues across a wide variety of system configurations and uptimes and therefore it is clearly necessary to introduce measures to ensure this data is acquired.

5.2. Hardware configuration

We used one bare-metal system for final correctness and performance testing our Volatility integration. It has a 6 core/12 thread Intel CPU, 64 GB of RAM, and runs Ubuntu 18.04.

We note that Volatility is not a multi threaded application so the number of cores has little impact on performance. The amount of RAM of the analyst's system can have a significant impact though as, if enough RAM is available, then an entire sample will be cached in memory once it is accessed. This will save a significant number of disk accesses and substantially improve performance compared to systems with smaller amounts of RAM.

6. Page state statistics

Before leveraging *vm_page_buckets* to aid in address translation, we first wanted to understand the state and purpose of pages that are actively tracked by the buckets. Through use of the previously mentioned *mac_walkqs* plugin, we were able to determine the relationship between RAM size and the number of tracked pages, how many pages are in a state suitable for use in address translation, and the distribution of pages across the various queues. Table 2 lists the distribution of page states for each sample in our data set. The number of pages in the file cache are separated from the other recoverable pages as their exact purpose is known just from being in that queue. Unfortunately, identification of pages in the file cache is only possible starting with Sierra so the samples from older versions have the file cached marked as *N/A*. Pages in the file cache queue always correspond to portions of cached files from disk, regardless of whether any processes are actively mapping the file. The pages that are in a recoverable state, but not in the file cache, are counted in the *Other* column. This includes pages belonging to anonymous memory, stacks, heaps, and other non-file backed regions. Only a relatively small number of pages were kept in the various free page queues, which constitute the *Invalid* column count. Analysis of the distribution showed that a significant percentage of physical pages in each sample were tracked in the page buckets and directly recoverable through examination of the corresponding *vm_page* instance.

7. Address space evaluation

After understanding the distribution of pages across physical memory, we then wanted to test and leverage our newly developed address space to examine process memory. We started with examination of two process memory regions critical to memory analysis. This was performed to determine precisely how much of previously unavailable critical process data could be recovered through analysis of the queues. Our first experiment was the examination of the stacks and heaps. We then deeply examined the recovery of the contents of process executables.

7.1. Recovery of stack and heap data

The process runtime stack holds metadata and data related to function calls. Depending on the compiler, programming language used, and the calling convention, this information can include return values, return addresses, function parameters, and local variables to functions. Analysis of stack data has a long history in memory forensics to uncover call stacks, parameters passed to sensitive APIs, and more. A process' heaps are what hold the dynamically generated data processed by the application and its associated libraries, frequently including artifacts such as keystrokes, images viewed or downloaded, HTTP requests and responses, copy/paste buffers, and other valuable forensic artifacts.

Due to the forensic value of the contents of stacks and heaps, we wanted to determine how many pages belonging to these regions could be recovered through the use of the queues. To start this analysis, we developed a new Volatility plugin that walked each process' set of memory mappings and then filtered for only those related to stacks and heaps and only those of 50 MB or less. The 50 MB restriction is to eliminate issues with data smear. It then attempted to translate the virtual address of each page within these regions and kept a count of how many pages translated to a physical offset and how many did not. This plugin was run twice against every sample in our data set. The first run was with an unaltered version of Volatility, and the second was with a version that had our new address space installed. Table 3 documents the result of this experiment. As illustrated, a substantial number of critical pages are made available through analysis of the queues.

7.2. Process executable recovery

Our second test involved attempting to recover the process executable for every running process in the data set of memory samples. This is a very common operation during analysis as recovering the executable from process memory provides several extremely beneficial advantages, but there are also several impediments to successful recovery.

7.2.1. Benefits and impediments

The benefits to process executable recovery include, but are not limited to, the following:

- **Defeating Packers** Upon execution, unpackers will usually decrypt/decompress/de-obfuscate strings and code from disk into their plaintext forms in memory. Recovery of these plain

Table 2
Physical page queue distribution.

#	File	Other	Invalid
1	N/A	229,591	364
2	N/A	396,138	1568
3	N/A	400,618	1001
4	34,674	214,979	693
5	37,581	446,832	3
6	67,574	387,421	14
7	27,874	460,329	12
8	88,104	686,018	0
9	361,541	1,976,958	4386
10	747,269	14,138,592	103,475

Table 3
Analysis of stacks and heaps pages.

#	Total	Valid without AS	Valid with AS
1	231,508	8248	11,565
2	452,008	50,024	54,849
3	442,254	61,366	66,679
4	588,739	59,538	76,055
5	518,504	66,299	68,248
6	220,230	24,458	25,533
7	776,664	69,889	79,377
8	2,999,337	106,712	220,213
9	11,216,197	49,750	479,436
10	17,411,435	17,604	383,827

text versions saves substantial reverse engineering time and effort (Ligh et al., 2010).

- **Recovering Injected Code** Code injecting malware replaces legitimate code in the address space of running processes with malicious alternatives. These changes to the code are not reflected back to the file on the file system and require memory forensics to detect. Common examples of such techniques are API hooks (Ligh et al., 2014) and process hollowing (Monnappa, 2016).
- **Recovery of Command and Control Data** Malware will often dynamically load configuration data from a remote command and control server. These configuration options can include a list of servers to exfiltrate information to, commands to run on the local system, and more. Such data is not available in the executable on the file system, but is available in the address spaces of actively executing malware.

These benefits cannot be fully realized when any of the impediments that hinder the ability to fully recover the executable or to recover it at all are present. The largest issue is that, to fully and properly recover an executable, its metadata must be available. This metadata encodes the layout and size of all of the file's sections and tells the memory forensics framework how to reconstruct the executable. The majority of this metadata is in the file's header, which will be mapped into the first page of the executable's memory region.

Unfortunately, since the header is never or rarely referenced after initial process loading, it is often a target of the swap manager since it will be not have been recently accessed. This will then remove the header's page from the process' address space, and with the header gone the memory forensics framework will produce an empty file when attempting to recover the executable.

Even when the header is present, analysis is still hindered in situations where many of the executable's pages are not present. Particularly in situations where an analyst wants to perform full static analysis of an executable from memory, it is very important to recover as much original data as possible as most reverse engineering tools are intolerant of corrupted executables.

We note that these issues related to recovering process executables affect analysis of all operating systems and not just macOS.

7.2.2. Executable headers and the queues

Our goal in testing process executable recovery was two-fold. First, we wanted to test whether incorporation of the queues could allow for recovery of the executable header for more processes than traditional translation provided. Second, we wanted to test how much more data inside of executables we could recover by processing the queues. To recover process executables, we relied on the existing *mac_procdump* Volatility plugin, which examines Mach-O files in memory and uses the metadata to properly reconstruct each section. We then ran the plugin twice as with *mac_walk_stack_heap*, once in an unaltered version of Volatility and once in a version with our address space installed.

The results of this experiment showed that there were 1893 active processes in total across our data set. Without our address space being active, 749 of these were recovered and 1144 were written as empty files. This produced 419 MB of data. With our address space active, 1366 executables were recovered and only 527 were written as empty files. This produced 943 MB of data in total. Table 4 lists the number of processes recovered with and without our address space active for each sample.

Table 4
Number of processes recovered.

#	Active Processes	Without AS	By AS
1	46	45	1
2	85	84	1
3	90	90	N/A
4	112	103	1
5	116	115	1
6	56	53	3
7	128	116	10
8	240	127	104
9	466	12	238
10	554	4	259

As shown in the table, a substantial number of previously unavailable executable headers become available with the use of our new queue-aware address space. This is exemplified in samples 9–10 as they were acquired on long running systems with substantially more processes and real-world work loads, all of which contribute to headers being made not-present over time by the swap manager. These results show that our developed address space has the potential for significant real-world benefits during investigations.

7.2.3. Executable data recovery with the queues

As mentioned previously, it is not just the headers of executables that can be made not present, but also any other portion of the executables as well. To maintain alignment, memory forensics will fill these missing pages with zeroes (NULL bytes). This can have severe, negative effects on the reverse engineering process as well as other types of forensics analysis. To further test the usefulness of our address space, we calculated how many pages of data belonging to executables were successfully recovered by *mac_procdump* in both runs. Table 5 lists the number of pages in executables that *mac_procdump* without our address space accessed as it attempted to recover executables. The table also lists the number of those pages that were previously non-present, but made present (recovered) by use of our address space. Processes that produced zero byte files without our address space, but that were recovered with our address space, are not included in this table's calculations as all of those pages were missing in the output without the queue analysis support.

As shown in the results, our address space provided the ability to recover a substantial number of pages belonging to process executables that otherwise would have been missed. Through use of our address space, analysts will be provided with much richer and complete data sets and artifacts to perform their investigations.

7.2.4. Performance impact

To assess the performance impact of our address space on analysis, we timed the execution of the previously described *mac_procdump* based analysis as well as monitored memory usage during the plugin's execution. The memory usage is based on the caching the address space does to avoid repeated calculations related to enumerating process mappings and object/offset

Table 5
Process Executable Data Recovered by our AS.

#	Total	Recovered	Recovered %
1	3679	908	24.68
2	11,325	3260	28.79
3	12,251	2584	21.09
4	12,181	4064	33.36
5	13,857	4958	35.78
6	6340	481	7.58
7	15,522	1895	12.20
8	20,238	3647	18.02
9	11,613	271	2.33
10	102	66	64.71

Table 6
Runtime of mac_procdump in seconds.

#	Runtime Without AS	Runtime with AS
1	20.25	26.24
2	31.14	40.54
3	38.88	40.46
4	32.99	42.18
5	32.02	40.74
6	22.89	30.27
7	23.94	31.47
8	27.88	61.40
9	21.13	277.02
10	19.01	549.46

calculations and lookups. Even on the large samples, these caches combined for only a few MBs of extra memory usage. Table 6 lists the runtime for *mac_procdump* with and without the address space installed.

Of course this minimal performance hit is far outweighed by the forensic information that is gained. The only samples that experienced substantial runtime spikes were samples #9 and #10. As shown previously in Table 4, sample #9 is the sample for which the address space recovered a further 238 processes compared to the non-queue aware Volatility. As a result of these 238 extra processes, the amount of data produced by the plugin for that sample grew from 46 MB to 234 MB. For sample #10, a further 259 processes were recovered, and the amount of data recovered went from 444 KB (kilobytes) to 291 MB. The few extra minutes these invocations take to run are well worth the substantial amount of extra forensic data.

8. Conclusion

Memory forensics will continue to be a critical technique in the fields of digital forensics, incident response, and malware analysis. Given the rise of memory only malware and exploits across all platforms, there is a strong need for memory forensics to recover as much structured data as possible from analyzed samples. In this paper, we have documented our effort to incorporate analysis of the macOS page queues into memory forensics. We chose Volatility as our research framework given its popularity in the industry as well as academia. The result of our work is the seamless integration of page queue analysis into all existing and future macOS plugins. As demonstrated in the results of analysis of stacks, heaps, and process executables, a substantial number of pages that were previously inaccessible are now made readily available. This will allow investigators to gather a much richer and more complete set of data and artifacts to perform thorough analysis. We plan to submit our new address space and associated research plugins to the Volatility project upon publication of this paper.

Acknowledgements

This work was supported in part by the National Science Foundation through grant # 1703683.

References

Amit, Singh, 2006. Mac OS X Internals: A Systems Approach. Addison-Wesley Professional.

Apple, 2020. Apple GitHub. https://github.com/apple/darwin-xnu.

Apple, 2020. Memory and Virtual Memory. https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/KernelProgramming/vm/vm.html.

Apple, 2020. Apple open source. https://opensource.apple.com.

Case, A., Richard III., Golden 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).

Cohen, Michael, 2014. Windows virtual address translation and the pagefile. http://blog.rekall-forensic.com/2014/10/windows-virtual-address-translation-and.html.

Dolan-Gavitt, B., 2007. The VAD tree: a process-eye view of physical memory. In: Proceedings of the 2007 Digital Forensic Research Workshop.

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

Gorman, Mel, 2014. Describing physical memory. https://www.kernel.org/doc/gorman/html/understand/understand005.html.

hdm, 2020. Metasploit. https://www.metasploit.com/.

Iqbal, Hameed, 2009. Forensic Analysis of Physical Memory and Page File. Master's thesis. Gjøvik University College.

Jared, M., 2008. Stimson. Forensic Analysis of Window's Virtual Memory Incorporating the System's Pagefile. Master's thesis. Naval Postgraduate School.

Jesse, D., 2007. Kornblum. Using every part of the buffalo in windows memory analysis. Digit. Invest. 4 (1), 24–29.

Karvandi, Sina, 2018. Inside windows page frame number (PFN) —— Part 1. https://rayanfam.com/topics/inside-windows-page-frame-number-part1/.

Levy, Jamie, 2015. Using mprotect(.., .., PROT_NONE) on Linux. https://volatility-labs.blogspot.com/2015/05/using-mprotect-protnone-on-linux.html.

Ligh, Michael, Adair, Steven, Blake, Hartstein, Richard, Matthew, 2010. Malware Analyst's Cookbook and DVD: Tools and Techniques for Fighting Malicious Code. Wiley Publishing.

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.

Monnappa, K.A., 2016. Detecting deceptive hollowing techniques. https://cysinfo.com/detecting-deceptive-hollowing-techniques/.

Mudge, Raphael, 2020. Cobalt Strike. https://www.cobaltstrike.com/.

Petroni, Nick, Walters, Aaron, Fraser, Timothy, Arbaugh, William, 2006. FATKit: a framework for the extraction and analysis of digital forensic data from volatile system memory. Digit. Invest. 3 (4).

PowerShell Empire, 2016. Empire: Building an Empire with Power Shell. https://www.powershellempire.com.

The Volatility Framework, 2017. Volatile Memory Artifact Extraction Utility Framework. https://github.com/volatilityfoundation/volatility.

Volatility Foundation, 2014. VMware snapshot file. https://github.com/volatilityfoundation/volatility/wiki/VMware-Snapshot-File.

Volexity, 2020. Surge Collect Pro. https://www.volexity.com/products-overview/surge/.

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