In lieu of swap: Analyzing compressed RAM in Mac OS X and Linux

Article *in* Digital Investigation · August 2014

DOI: 10.1016/j.diin.2014.05.011

2 authors:

Golden G. Richard III
Louisiana State University
91 PUBLICATIONS **1,743** CITATIONS
SEE PROFILE

Andrew Case
21 PUBLICATIONS **411** CITATIONS
SEE PROFILE

In Lieu of Swap: Analyzing Compressed RAM in Mac OS X and Linux

By

Golden Richard and Andrew Case

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2014 USA

Denver, CO (Aug 3rd - 6th)

Digital Investigation 11 (2014) S3–S12

Contents lists available at ScienceDirect

Digital Investigation

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

In lieu of swap: Analyzing compressed RAM in Mac OS X and Linux

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

[a] Department of Computer Science, University of New Orleans, New Orleans, LA 70148, USA
[b] Volatility Foundation, New Orleans, LA 70001, USA

Keywords:
Memory analysis
Live forensics
Compressed RAM
Virtual memory
Digital forensics

A B S T R A C T

The forensics community is increasingly embracing the use of memory analysis to enhance traditional storage-based forensics techniques, because memory analysis yields a wealth of information not available on non-volatile storage. Memory analysis involves capture of a system's physical memory so that the live state of a system can be investigated, including executing and terminated processes, application data, network connections, and more. One aspect of memory analysis that remains elusive is investigation of the system's swap file, which is a backing store for the operating system's virtual memory system. Swap files are a potentially interesting source of forensic evidence, but traditionally, most swap file analysis has consisted of string searches and scans for small binary structures, which may in some cases be revelatory, but are also fraught with provenance issues. Unfortunately, more sophisticated swap file analysis is complicated by the difficulty of capturing mutually consistent memory dumps and swap files, the increasing use of swap file encryption, and other issues. Fortunately, compressed RAM facilities, such as those in Mac OS X Mavericks and recent versions of the Linux kernel, attempt to reduce or eliminate swapping to disk through compression. The storage of compressed pages in RAM both increases performance and offers an opportunity to gather digital evidence which in the past would have been swapped out. This paper discusses the difficulty of analyzing swap files in more detail, the compressed RAM facilities in Mac OS X and Linux, and our new tools for analysis of compressed RAM. These tools are integrated into the open-source Volatility framework.

Introduction

Traditionally, digital forensics has focused primarily on non-volatile storage devices and involved preservation, imaging, recovery, and analysis of files stored on hard drives, removable media, etc. That investigative model typically embraced a "pull the plug and image" strategy, which involved powering down forensic targets without regard for their live state and making copies of non-volatile storage devices for analysis. This resulted in loss of a significant amount of potentially actionable digital evidence, including information about currently executing processes, live network connections, data in the clipboard, volatile malware, and other OS and application data structures. Increasingly, the forensics community has become aware of the potential for *live forensics* and *memory analysis* to enhance the investigative process, yielding evidence not available on non-volatile storage. Live forensics typically involves a survey of a running machine "on-the-spot", using a set of statically compiled binaries which are executed on the target to glean information about its state and available evidence. These tools are often traditional systems administration tools, which list running processes, monitor filesystem activity, capture network traffic, monitor changes to the Windows registry, and attempt to

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

http://dx.doi.org/10.1016/j.diin.2014.05.011
1742-2876/

detect malware, such as keystroke loggers. Memory analysis typically involves capture of a system's physical memory (e.g., a RAM dump, acquired via a combination of software and/or hardware) for later investigation, offline, using memory analysis tools. Live forensics and memory analysis are similar in that they both potentially offer a wealth of data to a forensic investigator that would be otherwise unavailable. Similarly, they are both potentially invasive, disturbing the state of a running system to varying degrees, but memory analysis strives to minimize this disruption by requiring only that a memory dumping utility be executed on the system, rather than a number of evidence-gathering applications. Because of recent research advances in memory analysis, much of the live system state observable with live forensics can now be recreated in the lab from a physical memory dump. One aspect of memory analysis that remains elusive is investigation of the system's swap file, which is a backing store for the operating system's virtual memory system. Virtual memory is discussed in greater detail in Section Memory analysis for modern virtual memory systems, but briefly, the swap file is typically stored on disk and contains the contents of physical memory pages that have been swapped out due to high *memory pressure*, essentially, a shortage of RAM induced by running large numbers of or particularly memory-hungry applications. The swap file can therefore contain actionable evidence, but because the swap file can be large and is stored on slow, non-volatile media, capturing a *mutually consistent* copy of both RAM and the swap file while a system continues to execute is very challenging.[1] There are additional challenges in swap file analysis, which are discussed in detail in Section Swap files as a source of evidence, but a new virtual memory component emerging in modern operating systems, called *compressed RAM* or *compressed swap*, offers an opportunity to gather digital evidence which in the past would have been swapped to disk. After providing some additional background in the following sections on virtual memory systems and memory analysis, we discuss our newly developed plugins for the Volatility framework, which automatically identify and decompress compressed memory regions in both Mac OS X Mavericks and Linux, making this data available for analysis. We also discuss the results of a series of experiments, which offer insight into the quantity and quality of the additional evidence made available by our plugins.

Memory analysis for modern virtual memory systems

Virtual memory is an essential component of modern operating systems, providing a linear address space for processes and significantly simplifying memory management. Operating systems often include a paging mechanism in the virtual memory system, to allow the total size of the allocated memory regions of executing processes to exceed the size of physical RAM, by overflowing RAM into a swap file. Primitive versions of paging have existed since the Atlas system in the 1960s (Morris et al., 1967). In this paper, we focus on operating systems that fully support paging, although some operating systems, particularly those for mobile or embedded devices, do support virtual memory but either do not support paging at all (e.g., QNX) or support paging but without a swap file, bringing in read-only pages as necessary from files on non-volatile storage (e.g., iOS). On modern hardware, virtual memory is implemented using a combination of hardware and software, with most modern CPUs providing hardware support for virtual to physical address translation and tracking whether pages are resident in RAM. Access to non-resident pages results in a *page fault*, which is handled by the operating system, triggering one of a number of possible actions, including allocation or the page being swapped in.

Swapping must be minimized to avoid *thrashing* (Denning, 1968a), where pages are continuously moved to and from the swap file because of a critical shortage of RAM, and the resultant impact on performance. Part of the reason that excessive swapping has such a serious impact on performance is the disparity between disk bandwidth and memory bandwidth, which differ by orders of magnitude. To illustrate this disparity, consider the memory bandwidth of the high- performance Mac Pro, introduced by Apple in 2013, which peaks at 60 GB/s. This model also sports some of the fastest flash storage to date, but storage bandwidth still peaks at 1.2 GB/s. To maximize performance, modern operating systems employ sophisticated

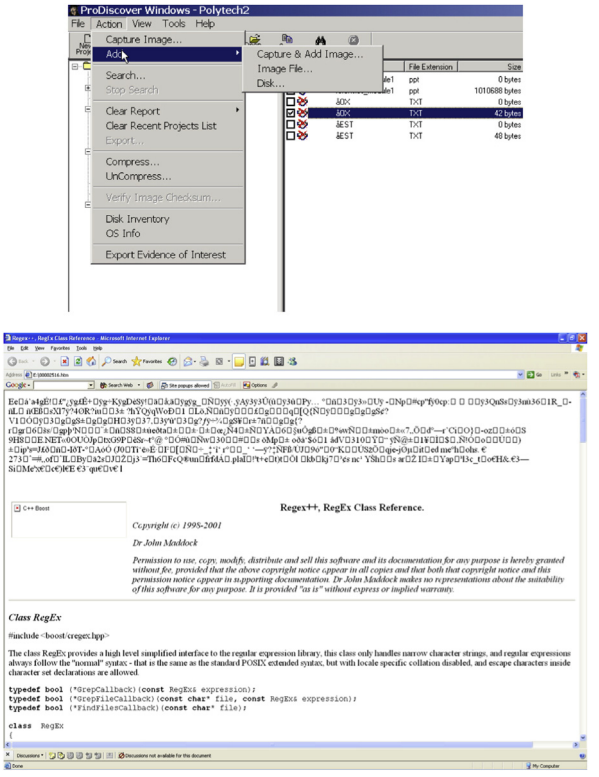

Fig. 1. Image file and HTML fragment carved from a Windows swap file. These are deleted documents "trapped" in the un-sanitized space allocated by Windows to the swap file.

[1] However, in virtualized environments, a virtual machine snapshot can be generated, which may reduce the level of inconsistency.

page replacement algorithms that strive to retain the pages that comprise the *working sets* (Denning, 1968b) for active processes, i.e., the pages that are in active use, in RAM, but some swapping is still inevitable when memory pressure increases. Swap files are discussed in more detail in the following section.

Swap files as a source of evidence

Swap files are a potentially interesting source of forensic evidence, but traditionally, most swap file analysis has consisted of searches for strings or small binary structures. Searches of this kind target web page fragments, passwords, credit card numbers, IP addresses (Garfinkel, 2013), etc. In many cases, matches in themselves are revelatory, but it is virtually impossible to establish the provenance of data in the swap file without analyzing operating systems kernel structures in the virtual memory system. First, the swap file is organized as an unordered collection of raw memory pages (or segments, which may exceed a page in size), and discerning even which process generated the data in the swap file independently of the kernel structures can't be done in a reliable way. Second, unless specific measures are taken, such as setting a registry key in Microsoft Windows,[2] the swap file is often not cleared when a system reboots, which leaves stale information in the swap file. This means that memory pages swapped out across (many) reboots may persist, all interleaved in the swap file. Third, and perhaps most importantly, some operating systems, such as Microsoft Windows, don't sanitize disk blocks when they are initially allocated to the swap file, meaning that data *completely unassociated with virtual memory can be present in the swap file*. An example of this is illustrated in Fig. 1, where file carving using Scalpel (Richard and Roussev, 2005) against a Windows swap file recovers an HTML file fragment and JPEG image "trapped" in the swap file, but unrelated to swapping activity.

An additional complication is the increasing use of encrypted swap files, to minimize the leakage of volatile, private information onto non-volatile storage. For example, in Mac OS X 10.7 and later, swap files are encrypted by default, regardless of whether File Vault 2 whole disk encryption is activated. Furthermore, the 128-bit AES keys for encrypting the swap files are distinct from those used for disk encryption, and are regenerated automatically every time the system boots. Consider the function `swap_crypt_ctx_initialize` (osfmk/vm/vm_pageout.c) in the Mac OS X Mavericks kernel (Apple, 2013), illustrated in Listing 1. This function is executed on the first page out operation after Mavericks is booted, ensuring that all pages swapped out during the current session will be encrypted using a new key. Support for encrypted swap is also available in Windows (for Vista and later), and in Linux, via the dm-crypt facility, although neither enables encrypted swap by default.

```
1  void swap_crypt_ctx_initialize(void) {
2    unsigned int i;
3    if (swap_crypt_ctx_initialized == FALSE) {
4       for (i = 0;
5          i < (sizeof (swap_crypt_key) /
6          sizeof (swap_crypt_key[0]));
7          i++) {
8            swap_crypt_key[i] = random();
9       }
10      aes_encrypt_key(
11         (const unsigned char *)swap_crypt_key,
12         SWAP_CRYPT_AES_KEY_SIZE,
13         &swap_crypt_ctx.encrypt);
14      aes_decrypt_key(
15         (const unsigned char *)swap_crypt_key,
16         SWAP_CRYPT_AES_KEY_SIZE,
17         &swap_crypt_ctx.decrypt);
18      swap_crypt_ctx_initialized = TRUE;
19    }
20 }
```

Listing 1. Definition of the `swap_crypt_ctx_initialize()` function in the Mac OS X kernel, which creates new encryption keys for the encrypted swap facility on the first page-out after a reboot.

Assuming that a dump of physical memory and a copy of the swap file are available, memory analysis can *theoretically* be used to associate swapped out pages in the virtual address spaces of processes with the owning processes and to deal with encrypted pages (by retrieving the keys from RAM). In practice, this doesn't work well, because of memory *smearing*[3] during capture, where the state of the process page tables and kernel structures governing virtual memory are inconsistent with the state of the swap file. This occurs because, except in virtualized environments where a virtual machine snapshot or virtual machine introspection (Javaid et al., 2012) can be used to more consistently capture or introspect both RAM and the swap file, it is generally very difficult in a running system to gather both in a consistent fashion (Kornblum, March 2007; Petroni et al., 2006). To see why, consider that while capture of the swap file is taking place, the system continues to execute, and mappings between pages, their owners, and locations in the swap file may change, resulting in significant inconsistency. If the recommended process of acquiring memory first is taken, then the analyst will acquire memory using hardware or software, and then use other software to acquire the swap file. On systems with moderate to large amounts of RAM, the initial memory acquisition can take several minutes. During this time the system is still executing programs and changing swap information. By the time both memory is sampled and the swap file is acquired from disk, it is very likely (and has been observed by the authors) that the kernel swap data structures will point to pages that have since been overwritten. If the acquisition tool blindly trusts this data then it will read data not actually associated with the translated virtual address. For example, it can mean a swapped-out mapping inside of Internet Explorer (under Windows) points to a swap file offset on disk that is actually now information from another

[2] For example, setting the ClearPageFileAtShutdown key in the registry instructs Windows to clear the swap file on shutdown, aimed at increasing security at the expense of performance.

[3] To our knowledge, Harlan Carvey coined the term smearing (Carvey, 2005), referring to inconsistencies in a physical memory dump as a result of the system continuing to execute during capture. In our view, the term is also descriptive of the potentially graver inconsistencies that occur between physical memory and the swap file when simultaneous capture is attempted.

process. The same issues occur even if an investigator attempts to acquire memory and the swap file from disk at the same time. Since there is no known way to programmatically detect these inconsistencies, swap file analysis is extremely difficult and furthermore, the validity of any conclusions based on the analysis is suspect.

The idea of compressing RAM to reduce memory pressure and thereby reduce or eliminate swapping isn't a new one, with commercial products such as RAM Doubler appearing around 1995. Academic research in this area is also well over a decade old (Wilson et al., 1999; Douglis, 1993). Some of the impediments to successful deployment of early RAM compression schemes were slow processors and poor system integration and tuning. The new compressed RAM facility in Mac OS X Mavericks and compressed swap facility in Linux, however, take advantage of fast multicore processors and tightly integrate compressed RAM into their virtual memory systems. By setting aside a moderate amount of RAM for a component we'll generically refer to as the *compressor* (see Section Analyzing compressed RAM for specific details on the Linux and Mac OS X implementations) and compressing memory pages and storing them instead of swapping them to disk, moderate memory pressure can be accommodated without swapping. When memory pressure increases beyond a certain threshold, then pages from RAM (and from the compressor) can still be swapped to disk.

With proper tools, compressed RAM is far easier to analyze than swap. Virtual memory smearing is minimized, because RAM can be captured much more quickly than swap, which resides on slower, non-volatile storage devices. To capitalize on this possibility, we have developed a set of tools to analyze compressed RAM, automatically decompressing and integrating compressed pages into the address spaces of processes under investigation. The next section provides details on the implementation of compressed RAM in Mac OS X and Linux and discusses our new analysis tools, which are incorporated into the Volatility framework (The volatility framework).

Analyzing compressed RAM

The Volatility Framework is a portable, open source framework for memory forensics, implemented entirely in Python. Volatility can analyze memory dumps from Linux, Mac, and Windows systems, among others, in both 32-bit and 64-bit flavors. A variety of memory dump formats are supported, including raw physical memory dumps, VMware snapshots, output from the Linux Memory Extractor (Sylve et al., 2012), and more. Volatility can be used to quickly and accurately model operating system kernel structures and handle virtual-to-physical-address translation and it also includes basic support for reconstructing virtual address spaces for processes, by noting which pages are present in the physical memory dump and which are not (e.g., pages that have been swapped out). The current release of Volatility does no processing for swapped out pages, aside from noting their absence. Particularly appealing is Volatility's flexible

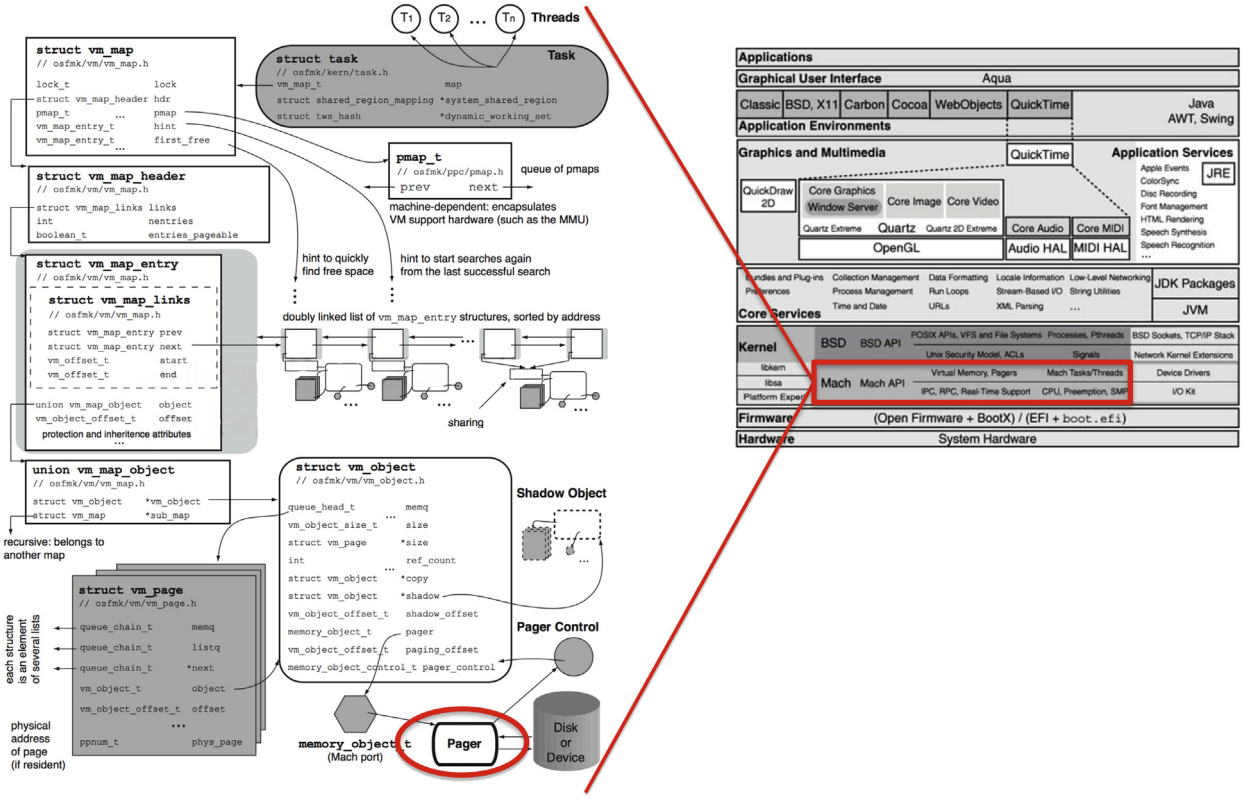

Fig. 2. Overview of the Mac OS X virtual memory system, which resides inside the Mach portion of the OS X kernel. A new pager, called the compressor pager, encapsulates most of the details of page compression/decompression. Adapted from Fig. 8-6 in Singh (2006).

plugin system, which allows the depth of its memory analysis capabilities to be expanded with little or no modification to the core system. More than 150 plugins are currently available.

To support the analysis of compressed RAM in Volatility for Mac OS X and Linux, we developed four new plugins: *mac_compressed_swap*, *mac_dump_maps*, *linux_compressed_ swap*, and *linux_dump_maps*, in addition to Python implementations of the necessary decompression techniques. Our goal in developing the plugins is broaden the scope of Volatility's virtual address space abstractions to include pages that are compressed (but not swapped out to disk). This means that any plugin that analyzes the address space of a process should encounter not only pages that are present in physical memory (signified by the present bit being set in the page table entry for the page), but also, transparently, pages that are currently not present but compressed. We discuss the implementations of the plugins in the following two sections. Experimental evaluation of the plugins, as well as performance issues, are discussed in Section Evaluation.

Compressed RAM in Mac OS X

Mac OS X employs a hybrid kernel, containing both BSD and Mach components. The virtual memory system is implemented primarily inside the Mach portion, as illustrated in Fig. 2. Each task (process) in Mac OS X has a flat address space represented in the kernel by a `vm_map`, which includes a doubly-linked list of `vm_map_entry` objects, accessible via the `vm_map_header` for the `vm_map`. Each of the `vm_map_entry` objects represents a set of contiguous memory addresses in the linear address space of the task and has an associated `vm_object`. The `vm_object` manages a set of 4 K pages, identified by their offset from the start of the object. These pages may be either resident in memory or retrievable from a *backing store*, which might be a file on disk (in the case of a memory-mapped file) or the swap file, for pages that have been swapped out, or in Mavericks, via decompression. A *pager* is responsible for retrieving pages that aren't present from the backing store. In Mavericks, two of the most important pagers in terms of memory analysis are the *vnode pager*, which manages pages backed by a file on disk (e.g., an executable) and the *compressor pager*, which arranges for transparent compression and decompression of pages that would otherwise be swapped to or from the swap file on disk.

One issue that's critical to understanding the implementation of compressed swap in Mac OS X is while each `vm_map_entry`'s `vm_object` representing a range of pages that can be swapped out has its own associated (private) pager (namely, an instance of the *compressor pager*), there is a single, global `compressor_object` that stores all the compressed pages and is actually responsible for compression and decompression operations. The compressor is also singly responsible for management of the compressed pages in its pool of memory, and periodically compacts the segments storing compressed pages for more efficient storage. The compressor also monitors available space for compressed pages and marks segments that should be swapped out to disk.

Volatility support

As a first step, to test the viability of identifying and decompressing compressed pages in Mac OS X, we concentrated solely on investigation of the compressed page store, which resides in the single, global `compressor_object`, defined in "osfmk/vm/vm_object.h". The management of the `compressor_object`'s storage and high-level compression and decompression functionality is implemented in "vm_compressor.h" and "vm_compressor.c" (osfmk/vm/), with hand-optimized 64-bit assembler versions of WKdm compression/decompression implemented in "WKdmDecompress_new.s", "WKdmData_new.s", and "WKdmCompress_new.s" (osfmk/x64_64/). We analyzed the assembler implementations of WKdm, which consist of approximately 1000 lines of 64-bit assembler, and found them to be relatively similar to the original WKdm implementation described by Wilson and Kaplan (Wilson et al., 1999).

In order to decompress pages in Python (required by Volatility), we implemented a Python version of WKdm, based on the assembler version for compatibility. Next, we implemented the *mac_compressed_swap* plugin for Volatility, which extracts the locations of important kernel data structures related to the compressor from the physical memory dump, outputs statistics, including the total amount of memory available, number of compressed pages, etc. and then analyzes the main compressor store, which holds compressed pages. Simplifying slightly, this store is organized as a dynamically allocated array of structures of type `c_segment`, depicted in Listing 2. Each of these `c_segment` structures tracks information about a collection of compressed pages, stored in a single contiguous buffer per segment (accessible via the field `c_buffer`). A bitfield in the `c_segment` tracks a number of important characteristics of the segment, including whether it's currently swapped out.

```
1  struct c_segment {
2  ...
3     int32_t     c_bytes_used;
4     int32_t     c_bytes_unused;
5     uint32_t    c_mysegno:19,
6  ...
7                 c_ondisk:1,
8                 c_was_swapped_in:1,
9                 c_on_minorcompact_q:1,
10                c_on_age_q:1,
11                c_on_swappedin_q:1,
12                c_on_swapout_q:1,
13                c_on_swappedout_q:1,
14                c_on_swappedout_sparse_q:1;
15    uint16_t    c_firstemptyslot;
16    uint16_t    c_nextslot;
17    uint32_t    c_nextoffset;
18    uint32_t    c_populated_offset;
19    uint32_t    c_creation_ts;
20    uint32_t    c_swappedin_ts;
21    union {
22            int32_t *c_buffer;
23            uint64_t c_swap_handle;
24    } c_store;
25 ...
26    struct c_slot *c_slots[C_SEG_SLOT_ARRAYS];
27 };
```

Listing 2. Structure of one segment in the Mac OS X compressor's store.

The locations of pages in the buffer are tracked using a per-segment 2D array, which identifies the offset into the `c_buffer` where the page is stored and the length of the compressed page. The location of a compressed page in the compressor is identified by a *slot*, represented by a `c_slot_mapping` structure (illustrated in Listing 3), which tracks the segment number (`s_cseg`) and allows computation of the index into the array of `c_segment` structures and a 10-bit index (`s_cindx`), which is broken down into two smaller indices (using bit masks) into the 2D array of page locations for the segment. When a page in the address space of a process is compressed or decompressed, the mapping between its address and its location in the compressor is maintained by the pager associated with the `vm_map_entry` containing the page. The point of this is that the singleton compressor object doesn't track the virtual address or owning process for a compressed page — it simply operates on slots that track locations of arbitrary pages in its store. A higher level entity, namely, the pager, must be able to associate individual pages (by address) with their storage slots. The data structures responsible for mapping a virtual address to a compressed page are illustrated in Fig. 3.

```
1  struct   c_slot_mapping {
2          uint32_t          s_cseg:22,
3                            s_cindx:10;
4  };
```

Listing 3. Structure of the slot mapping structure, which allows the kernel to track the location of a compressed page.

In contrast to the Linux implementation of compressed swap, discussed in the next section, Mac OS X swaps out entire, compressed segments when the compressor runs low on space.

Our *mac_compressed_swap* plugin walks the array of `c_segment` structures, bounded by the value stored in the kernel variable `c_segment_count`, identifying segments that are present. For each segment marked present, we iterate over valid indices in the 2D slot array, and compute the location and length of the corresponding page in the `c_buffer`. A "compressed" page whose length is exactly 4 K was stored uncompressed, because WKdm compression failed to reduce its memory footprint — these pages are simply copied directly into output files for further analysis. Otherwise, the compressed data is fed into our Python WKdm decompressor and, if decompression succeeds, the decompressed page is output. Note that the format of compressed pages is sufficiently complex that "successful" decompression of corrupted data is very unlikely — our implementation of WKdm monitors decompression and aborts if errors are encountered, substituting a zero page. Note that this plugin does not attempt to associate decompressed pages with their owning processes — its goal is to simply dump all compressed swap for further analysis. This is because provenance information for compressed pages is not available in the compressor. Associating pages in a process's address space with their location in the compressor requires a mapping between the page's address and its slot in the compressor, which is handled by the pager

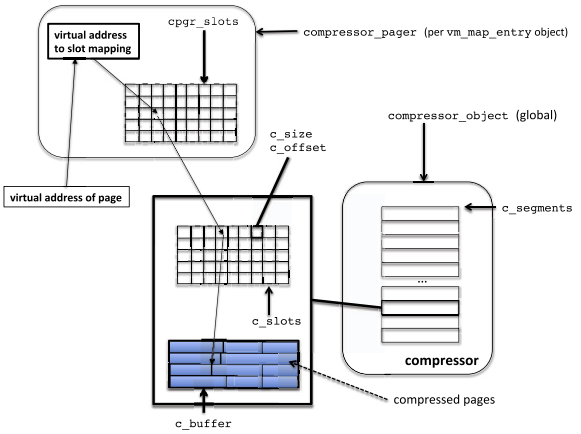

Fig. 3. High level structure of the compressor pager and compressor data structures, illustrating the internals of the highlighted "pager" in Fig. 2. A compressor_pager is associated with each vm_map_entry object, which tracks a set of contiguous pages in the virtual address space of a task, including the slots for compressed pages. The global compressor object stores and manages the compressed pages of all tasks.

object for a `vm_map_entry` object. We undertook that task in developing the *mac_dump_maps* plugin, discussed next.

Associating compressed data with individual processes is more complex, and requires a deep understanding of the entire Mach paging subsystem, which consists of approximately 70,000 lines of C. Since our preliminary work on decompression and analysis succeeded, we designed and implemented a second Volatility plugin, *mac_dump_maps*, which models a subset of the Mach VM page — in/decompression code path in Python. Our plugin walks the virtual address spaces of one or more processes, dumping all available pages in the address spaces (in order) to an output file. Ordinarily, Volatility's facilities for dumping process memory would skip pages that aren't present. Our plugin interjects a "second chance" for accessing a page's contents by first determining if the pager for the associated *vm_map_entry* is the compressor pager, and if so, querying the compressor pager to see if the page can be retrieved. If the page exists in the compressor, then techniques used for the previous plugin are used to decompress the page and make it available.

Compressed swap in Linux

Like Mac OS X, Linux (with kernel v3.11+) now offers a mechanism for handling memory pressure, called "compressed cache for swap pages" (hereafter, just "compressed swap"). The purpose of this feature is to allow transparent, in-memory compression of pages that normally would be swapped to disk. The use of compressed swap is optional and can be turned on by enabling the ZSWAP configuration option. There is also a kernel boot parameter, zswap.enabled, that controls whether zswap (compressed swap) will be used or not.

Frontswap

Frontswap is a generic API that implements compressed swap. In particular, it manages pages that are being

swapped out, loading of pages that are being swapped in, and the registering and unregistering of frontswap backends such as zswap, which is discussed next. Frontswap is integrated into the kernel via the `swap_writepage` and `swap_readpage` functions. These functions are called as pages are being written to or read from swap. Normally, these functions will directly read and write data from disk. When frontswap is enabled, though, the frontswap operation is tested before the disk being accessed, as illustrated in the `swap_writepage` code in Listing 4.

```
1  int swap_writepage(struct page *page,
2      struct writeback_control *wbc)
3  {
4      int ret = 0;
5      if (try_to_free_swap(page)) {
6          unlock_page(page);
7          goto out;
8      }
9      if (frontswap_store(page) == 0) {
10         set_page_writeback(page);
11         unlock_page(page);
12         end_page_writeback(page);
13         goto out;
14     }
15     ret = __swap_writepage(page, wbc,
16                 end_swap_bio_write);
17 out:
18     return ret;
19 }
```

Listing 4. The `swap_writepage()` function in the Linux virtual memory subsystem, which attempts to use the Linux frontswap facility to avoid writing a page to disk that's targeted for swap-out.

The function `swap_writepage` first calls `try_to_free_swap` which checks if the page still needs to be kept within swap storage. If this function returns a non-zero value, then line 9 executes and calls into the frontswap facility. If frontswap can handle the page (e.g., compress and store it), then the goto on line 13 is executed, and the write to disk on line 15 is never executed. `swap_readpage` operates in a similar manner and a page is checked for existence in the compressed cache before it is searched for on disk. `frontswap_store` is a simple wrapper that checks if frontswap is enabled and if so calls `__frontswap_store`. This function takes the page structure of the page to be stored as a parameter and performs the interaction with zswap. In order to ask zswap to store the page, it must be able to tell it the type of the page and its offset within the swap cache. This information is encoded within the `private` member of the associated `struct page`, which for swap files points to a `swp_entry_t` structure. A `swp_entry_t` encodes the type and offset of the page within its `val` member using bitmasks. The `type` bitmask specifies on which swap file the operation occurs, as Linux supports multiple swap files, and the `offset` is simply the offset within the particular file. Once these values are extracted, the store operation of zswap is called with the values and the offset of the page is marked as in-use within the frontswap bitmask. Listing 5 contains a stripped down version of `__frontswap_store` that illustrates these concepts. On lines 2–7 the `swp_entry_t structure` is extracted and the type and offset of the page calculated. On line 8, the store method of zswap is called with these parameters, and if the store was successful (a return value of 0), then the bit is set within the bitmap. `frontswap_load` performs the inverse operation of `frontswap_store`. When a page is requested from the compressed swap cache, `frontswap_load` extracts the type and offset from the page structure and then looks up the entry within zswap, using the load operation.

```
1  int __frontswap_store(struct page *page) {
2      swp_entry_t entry=
3        { .val=page_private(page),};
4      int type=swp_type(entry);
5      struct swap_info_struct
6          *sis=swap_info[type];
7      pgoff_t offset=swp_offset(entry);
8  ...
9      ret = frontswap_ops->store(type, offset,
10         page);
11     if (ret==0) {
12         set_bit(offset, sis->frontswap_map);
13 ...
```

Listing 5. The `__frontswap_store()` function extracts the type and offset of a page and then stores it within zswap.

Zswap

Zswap is an implementation of compressed swap pages for Linux. It is currently, as of 3.11, in the mainline kernel and is due for inclusion into new releases of several distributions. It implements the frontswap API by storing compressed pages and managing them through use of the zbud allocator and a tree of compressed pages per swap file. zbud (mm/zbud.c) is a memory allocator designed for compressed pages and stores two compressed pages within a single physical page, as well as managing the allocation, deallocation, and tracking the status of each page.

The per-swap file tree of compressed pages is represented as a `struct zswap_tree`. This structure holds a red-black tree of pages in its `rbroot` member, and a pointer to the zbud allocation pool in its `pool` member. Each node of the tree is stored as a `zswap_entry` structure, which holds the file offset, length of compressed buffer, and handle of the entry. The `handle` member is a pointer to the compressed data tracked by the tree node.

Of particular importance to memory forensics are the load and store implementations of zswap. In order to successfully find pages within the cache, the store operation and its associated data structures must be understood. `zswap_frontswap_store` is zswap's implementation of the frontswap store operation and is responsible for a number of tasks:

1. Compressing a given page
2. Allocating zbud memory to store it
3. Creating and populating a `zswap_entry` with the information
4. Storing the `zswap_entry` within the tree of pages
5. Updating statistical information that can be used to monitor the overall state of zswap

The default compression algorithm is LZO, which is implemented in the `lzo1x_compress_safe` and `lzo1x_decompress_safe` functions in the kernel. The compression algorithm can be changed at boot time through use of the `zswap.compressor` kernel parameter. The current implementation of our plugins only supports LZO, but adding support for other algorithms would simply require a Python port of the new decompression algorithm. Unlike Mac OS X, which utilizes a hand-optimized assembler implementation of WKdm for performance, Linux uses a straightforward C implementation for all the compression code we've studied. This makes the addition of new algorithms straightforward.

`zswap_frontswap_load` is zswap's implementation of the frontswap load operation. This function looks up a specific page in the tree and then decompresses it. This is the same operation that our Volatility plugins must perform when they decompress pages in the compressed swap cache.

Volatility support

Our first Linux plugin is *linux_compressed_swap*, which outputs statistical information on the state of frontswap and zswap and also extracts all compressed pages to disk in their uncompressed form. The second plugin we developed is a completely rewritten version of *linux_dump_maps*, which writes memory mappings to disk inclusive of the decompressed pages.

linux_compressed_swap first gathers information from zswap and frontswap, such as the global number of compressed pages as well as number of compressed pages for each swap file. This information is displayed and then the plugin inspects each swap file's zswap tree and locates every compressed page. This is accomplished by using the `rbroot` member and traversing the tree. Each tree element is an `rb_node` structure that is then converted to a `zswap_entry`. From there, the `handle` and `length` members can be used to find the compressed buffer. This buffer is then passed to the decompression routine and written to disk.

linux_dump_maps operates by walking the memory mappings of each process and extracting them to disk. These mappings can include the executable file, shared libraries, stack, heap, and anonymous memory of the process. In the current version of Volatility (2.3.1), the code leverages Volatility's API to find the starting and ending virtual address of each memory region of a process. It then walks each region, 4 K bytes at a time. If the page is in RAM it is written to disk. If the page is not in RAM, then the page is filled with zeroes and written to disk in order to maintain alignment and offsets. In our rewrite of this plugin, we instrument the reading process so that if a page is not found in RAM, then the compressed swap cache is searched to locate the page. This checking occurs by first finding the page table entry (PTE) of the non-present page and extracting its `type` and `offset`. This mimics the operation performed by `zswap_frontswap_load`. The offset is then checked against the frontswap bitmap of page offsets. If the corresponding bit is set then the page is backed by the cache. The `zswap_entry` of the page is then discovered by walking the tree of pages and finding the node that has the same offset as the one calculated for the page. This is how the tree is keyed and guarantees uniqueness of each node. Once the correct `zswap_entry` is found, its pages can be extracted by re-using code that was developed for the *linux_compressed_swap* plugin.

Evaluation

In designing, implementing, testing, and evaluating the potential impact of our tools, we analyzed a number of Mac OS X and Linux configurations, varying both the total amount of RAM available and memory pressure. Our concerns were both correct operation and a sense of how the inclusion (or exclusion) of compressed regions in memory analysis might affect an investigation, both qualitatively (evidence available only in the compressed regions) and quantitatively (how much data is actually compressed). Of course the quantity and relevance of evidence in compressed RAM will vary on a case-by-case basis, but our conclusion, based on experience with a large number of systems with between 2 GB and 32 GB of RAM, is that a substantial amount of data may be compressed and inaccessible to tools that are agnostic to compressed RAM. This is particularly true of Mac OS X, where compressed RAM is used aggressively. The Activity Monitor application tracks memory pressure and provides real-time stats on the compressor, as illustrated in Fig. 4, where on a system with a modest amount of RAM, almost 1 GB is compressed, with no swapping at all. Below, we provide results for a Mac and a Linux case, representative of our experience. Both are for systems with 2 GB of RAM, running modest workloads. On average, more data will be compressed on systems with less RAM, but as we discussed above, we have observed significant compressor activity even on systems with 32 GB of RAM, while running realistic workloads.

For our representative Mac case, we targeted a 64-bit Mac OS X Mavericks system with 2 GB RAM and 124 processes executing, and relatively high memory pressure. The compressed page pool at the moment we dumped RAM contained approximately 300 MB of data. We reviewed the

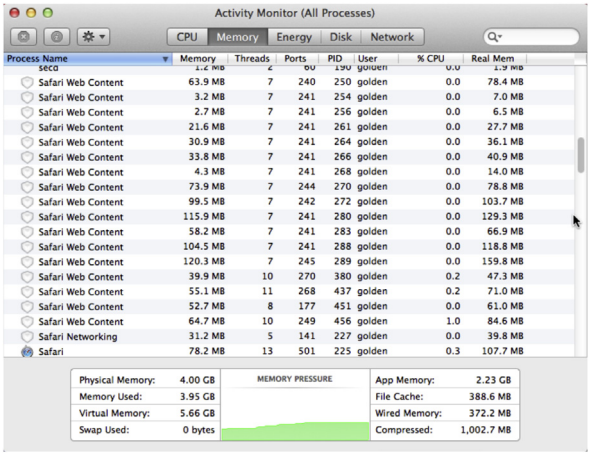

Fig. 4. : Activity Monitor in Mac OS X Mavericks, illustrating memory pressure and use of compressed RAM. In this example, a number of Safari windows are open, resulting in over 1 GB of compression and no swapping at all.

number of compressed pages for each process, and determined that 21 had no compressed pages at all, 86 had between 1 and 1000 compressed 4 K pages, 15 processes had between 1001 and 10,000 compressed pages, and 2 processes had more than 20,000 compressed pages.

For our representative Linux case, we targeted a 64-bit Ubuntu system with 2 GB of RAM, 115 processes executing, and relatively high memory pressure. We took a physical memory dump and with our tools, observed that the compressor held 160 MB of compressed data, 46 processes had no compressed pages, 66 had between 1 and 1000 compressed 4 K pages, and the rest between 2000 and 13,000 compressed pages.

In the tests above, the user was browsing (e.g., looking at images on Flickr), editing several large text documents (which were not saved on disk), and performing other routine computing tasks. When we analyzed the compressed portions of the process address spaces, we found substantial amounts of evidence, including large web page fragments identifying the pictures being viewed, recoverable thumbnail images, large chunks of the text documents, and more. This information would have been inaccessible without our tools.

One drawback in using the Volatility framework for compressed RAM analysis is performance, since Python is a poor choice for computationally intensive operations such as decompression. On Mac OS X, the compression and decompression functions are written in optimized, 64-bit assembler and on Linux, in C. We concentrated on the Mac OS X implementation of WKdm as a "reality check" for the performance of our Python implementation. In our evaluation, we consider compression of a 4 K page and subsequent decompression as a single "operation". On a Core i7 the WKdm functions in assembler are capable of >260,000 operations per second. An optimized C implementation is capable of >142,000 operations per second on the same system. The Python implementation achieved only 390 operations per second. While memory analysis is an offline operation, clearly for computationally intensive forensic analysis, pure Python is a poor choice, despite the other obvious advantages of implementing Volatility (and other tools) in Python. Python is currently mandated by Volatility for portability reasons.

Related work

There has been a substantial amount of high-quality research in memory analysis since the 2005 DFRWS memory analysis challenge provided the catalyst for work in this area. Volatility (The volatility framework) has emerged as a primary framework for integrating new results in memory analysis for Linux, Windows, Mac, and Android systems, but a number of other research efforts have contributed substantially to the analysis capabilities that investigators now have at their disposal (Sylve et al., 2012; Carbone et al., 2009; Dolan-Gavitt et al., 2009; Schuster, 2006; Carrier, 2006; Adelstein, 2006; Case et al., 2008, 2010a, 2010b; Van Baar et al., 2008; Dolan-Gavitt, 2007; Vidas, 2007).

Little formal work exists on swap file analysis, due primarily (we suspect) to many of the problems we pointed out in Section Swap files as a source of evidence. As we noted there, some of our observations regarding the difficulties related to swap file analysis are due to Kornblum (Kornblum, March 2007) and Walters (Petroni et al., 2006). One recent research effort that targets swap file analysis in conjunction with traditional memory analysis is the work by Javaid et al., which analyzes process heap behavior to detect malware and uses virtual machine introspection to incorporate live swap file analysis (Javaid et al., 2012).

The Linux and Mac compressed swap facilities discussed in this paper are new and to our knowledge, we are the first researchers to evaluate the forensic value of these facilities and to present tools capable of analyzing compressed swap.

Conclusions

Whereas swap was previously difficult to integrate into forensics analysis, the new implementations of in-memory, compressed swap in Mac OS X and Linux make much of the data available to investigators using standard memory acquisition techniques. In this paper we documented the implementations of compressed swap on both Mac OS X Mavericks and Linux and described our new open source tools for decompressing and analyzing compressed memory regions. The tools, which are integrated into the Volatility framework, allow extraction of all compressed pages at once and for per-process extraction. The complete extraction of all pages can be very useful when performing searches for content like personally identifiable information (PII), financial information, or for malware with Yara signatures. The per-process extraction ensures that as much information as can be found in memory will be included in the process memory dumps. Our experiments suggest that in many circumstances, processing compressed memory will make the difference between the investigator being able to properly analyze a process (or not), as a substantial amount of a process's address space may be compressed. We also illustrated a number of useful forensics artifacts discovered in compressed pages, in routine testing, reinforcing our position that analysis of compressed swap is necessary.

Future work

We expect that compressed RAM will be integrated into other operating systems, such as Microsoft Windows, in the future, based on the positive reviews associated with the Mac and Linux implementations. Closed source operating systems will present new challenges, as modeling of substantial portions of the virtual memory system is necessary for decompressing memory. Supporting analysis of compressed swap on mobile devices is also part of our future work. For example, Android 4.4 optionally enables frontswap, and handling compressed swap on ARM-based devices will require updates to Volatility's core functionality (e.g., the page table entry walking code will require updates to support ARM).

Another aspect of our future work is creation of publicly available memory samples with compression enabled. When we began this project, there were no publicly available memory samples that we could use. To support

additional research in this area, we are in the process of creating memory samples that can be released to the entire community.

References

Adelstein Frank. Live forensics: diagnosing your system without killing it first. Commun ACM 2006;49(2):63–6.

Apple. Xnu source code http://www.opensource.apple.com/source/xnu/xnu-2422.1.72/; 2013.

Carbone Martim, Cui Weidong, Lu Long, Lee Wenke, Peinado Marcus, Jiang Xuxian. Mapping kernel objects to enable systematic integrity checking. In: Proceedings of the 16th ACM conference on Computer and communications security; 2009.

Carrier Brian D. Risks of live digital forensic analysis. Commun ACM 2006;49(2):56–61.

Carvey Harlan. Origin of the term "smear" in memory analysis http://seclists.org/incidents/2005/Jun/22; 2005.

Case Andrew, Cristina Andew, Marziale Lodovico, Richard III Golden G, Roussev Vassil. Face: automated digital evidence discovery and correlation. In: Proceedings of the 8th Annual Digital Forensics Research Workshop (DFRWS 2008); 2008.

Case A, Marziale L, Richard III Golden G. Dynamic recreation of kernel data structures for live forensics. In: Proceedings of the 10th Annual Digital Forensics Research Workshop (DFRWS 2010); 2010.

Case Andrew, Marziale Lodovico, Neckar Chris, Richard III Golden G. Treasure and tragedy in kmem-cache mining for live forensics investigation. In: Proceedings of the 10th Annual Digital Forensics Research Workshop (DFRWS 2010); 2010.

Denning Peter J. Thrashing: its causes and prevention. In: Proceedings of the 1968 Fall Joint Computer Conference, Part I. ACM; 1968a. pp. 915–22.

Denning Peter J. The working set model for program behavior. Commun ACM 1968b;11(5):323–33.

Dolan-Gavitt Brendan. The vad tree: a process-eye view of physical memory. Digit Investig 2007;4:62–4.

Dolan-Gavitt Brendan, Srivastava Abhinav, Traynor Patrick, Giffin Jonathon. Robust signatures for kernel data structures. In: Proceedings of the 16th ACM conference on Computer and communications security. ACM; 2009. pp. 566–77.

Douglis Fred. The compression cache: using on-line compression to extend physical memory. In: Proceedings of the USENIX Winter Conference; 1993. pp. 519–29.

Garfinkel Simson L. Digital media triage with bulk data analysis and bulk extractor. Comput Secur 2013;32:56–72.

Javaid Salman, Zoranic Aleksander, Ahmed Irfan, Richard III Golden G. Atomizer: a fast, scalable and lightweight heap analyzer for virtual machines in a cloud environment. In: Proceedings of the 6th Layered Assurance Workshop (LAW'12); 2012.

Kornblum Jesse D. Using every part of the buffalo in windows memory analysis. Digit Investig March 2007;4(1):24–9.

Morris Derrick, Sumner Frank H, Wyld Michael T. An appraisal of the atlas supervisor. In: Proceedings of the 1967 22nd National Conference, ACM '67. New York, NY, USA: ACM; 1967. pp. 67–75.

Petroni Jr Nick L, Walters Aaron, Fraser Timothy, Arbaugh William A. Fatkit: a framework for the extraction and analysis of digital forensic data from volatile system memory. Digit Investig 2006;3(4):197–210.

Richard III Golden G, Roussev Vassil. Scalpel: a frugal, high performance file carver. In: Digital Forensics Research Conference (DFRWS 2005); 2005. pp. 71–7.

Schuster Andreas. Searching for processes and threads in microsoft windows memory dumps. Digit Investig 2006;3:10–6.

Singh Amit. Mac OS X internals: a systems approach. Addison-Wesley Professional; 2006.

Sylve Joe, Case Andrew, Marziale Lodovico, Richard III Golden G. Acquisition and analysis of volatile memory from android devices. Digit Investig 2012;8(3):175–84.

The volatility framework: Volatile memory artifact extraction utility framework. http://www.volatilesystems.com/default/volatility.

Van Baar RB, Alink Wouter, Van Ballegooij AR. Forensic memory analysis: files mapped in memory. Digit Investig 2008;5:S52–7.

Vidas Timothy. The acquisition and analysis of random access memory. J Digital Forensic Pract 2007;1(4):315–23.

Wilson Paul R, Kaplan Scott F, Smaragdakis Yannis. The case for compressed caching in virtual memory systems. In: USENIX Annual Technical Conference, General Track; 1999. pp. 101–16.